LAHDEN AMMATTIKORKEAKOULU
Lahti University of Applied Sciences

LAITEOHJAUSKIRJASTON SÄIETURVALLISUUS

Moniajoa tukevan laiteohjauskirjaston toteutus UFG:lle

LAHDEN AMMATTIKORKEAKOULU
Tekniikan ala
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Opinnäytetyö
Syksy 2011
Tuomas Terho

Lahden ammattikorkeakoulu
Tietotekniikan koulutusohjelma

TERHO, TUOMAS: Laiteohjauskirjaston säieturvallisuus
Moniajoa tukevan laiteohjauskirjaston toteutus UFG:lle

Ohjelmistotekniikan opinnäytetyö, 75 sivua

Syksy 2011

TIIVISTELMÄ

Tämä opinnäytetyö käsittelee laiteohjauskirjaston suunnittelua ja käytännön toteutusta. Työssä käydään läpi eteen tulleita ongelmia ja niiden ratkaisuja, ympäristön asettamia vaatimuksia sekä toteuttamiseen käytettyjä tekniikoita. Työssä vastataan kysymykseen, mitä haasteita moniajo tuo kirjaston suunnitteluun. Lisäksi tarkastellaan, kuinka oliopohjaisella rakenteellisella ratkaisulla ja Windows-käyttöjärjestelmän suojausmenetelmillä voidaan toteuttaa säieturvallinen laiteohjauskirjasto.

Laiteohjauskirjastoa käytetään osana Unigraf oy:n UFG-04- ja UFG-06-tuoteperheiden kuvankaappauslaitteiden ohjelmistoa ja sovelluskehitysalustaa. Kirjastolla voidaan lukea laitteelta signaalin analysointitietoa, ohjata laitetta ja kaapata kuvaa, ääntä sekä tapahtumalokia.

Laiteohjauskirjastolla voidaan ohjata useita erilaisia laitteita yksikorttisista monikorttisiin ja erilaisilla videostandardeilla varustettuihin. Kirjaston ohjelmointirajapinta piilottaa laitteiden monimutkaisuuden ja -muotoisuuden käyttäjältä ja yhtenäistää laitteiden ohjauksen tarkasti rajatun funktiorajapinnan avulla.

Kehitystyön pääpaino oli oliopohjaisen ohjausjärjestelmän ja sen eri tasoilla tapahtuvan resurssien suojauksen toteuttamisessa. Laiteohjauskirjastoon toteutettiin yleiskäyttöinen kehysrakenne, joka soveltuu erilaisten laitteiden ohjaukseen ja sisältää suurimman osan laiteohjauksessa tarvittavasta resurssien suojauksesta.

Toteutettu laiteohjauskirjasto koostuu kehysrakenteesta, laitekohtaisista erityispiirteisistä luokista ja ohjelmointirajapinnasta. Rakenne on joustava ja sallii monenlaisten laitteiden liittämisen osaksi kirjastoa. Se helpottaa ja nopeuttaa tulevien tuotteiden kehitystyötä.

Laiteohjauskirjaston toteutuksessa päästiin haluttuihin tavoitteisiin ja kaikki asetetut vaatimukset täytettiin. Työssä toteutettu kirjasto on käytössä Unigrafin omissa ohjelmistotuotteissa ja on saanut kiitosta myös asiakkailta. Kirjastoa kehitetään jatkuvasti.

Avainsanat: C++, mutex, moniajo, resurssien suojaus, API, kuvankaappaus

Lahti University of Applied Sciences
Degree Programme in Information Technology

TERHO, TUOMAS: Thread-safety in a device control library
Implementation of a multitasking supporting device control library for UFG

Bachelor´s Thesis in Software Engineering, 75 pages

Autumn 2011

ABSTRACT

This study discusses designing and implementing a device control library. The study deals with requirements set by the environment and techniques used within the implementation, as well as problems encountered in the implementation, and the solutions. The study answers questions about what challenges the multitasking brings to the library design and how to realize a thread-safe device control library using an object-based approach, procedural design, and the resource protection methods of the Windows operating system.

The device control library is used as a part of Unigraf Oy's solutions and software development kit for UFG-04 and UFG-06 product families. The library is capable of reading signal analysis information, capturing image, audio and event log data, and controlling UFG devices.

The device control library can handle multiple devices from single cards to card arrays supplied with different video input standards. The application programming interface in the library hides complexity and diversity from a user and unifies the device control by a strictly defined function interface.

The main focus of the development process was in the object-based control system and implementing its resource protection on different levels. A general purpose framework was developed into the library which can be used to control different devices and which contains the most of the synchronization features needed for thread-safe device control.

The library consists of the framework, of specialized device-oriented classes and of the application programming interface. The construction is adaptive and enables many kinds of devices being supported by the library. It facilitates and speeds up the development of future devices.

All goals and requirements were fulfilled within the practical implementation of the device control library. The library implemented within this study is currently in use in Unigraf's software products and has also gained acknowledgement from customers. The library is under constant development.

Keywords: C++, mutex, multitasking, synchronization, API, frame grabbing

LYHENTEET

API	Application Programming Interface – Sovellusohjelmointirajapinta, jonka avulla ohjelmat voivat suorittaa toimintoja keskenään.
BAR	Base Address Registers – PCI-väylän muisti- ja I/O-osoiteavaruus.
CTS	Compliance Test Suite – Sarja standardointijärjestö VESA:n hyväksymiä video- ja multimedialaitteiden määräystenmukaisuustestauksen työkaluja.
DDR2	Double Data Rate SDRAM – Kaksinkertaisella nopeudella toimiva muisti, jossa dataa siirretään sekä kellon nousevalla että laskevalla reunalla.
DLL	Dynamic Link Library – Dynaamisesti linkattava kirjasto, sovelluslaajennus.
DMA	Direct Memory Access – Tapa siirtää nopeasti tietoa laitteelta muistiin tai muistista laitteelle. Käytetään joissakin tapauksissa myös siirroissa muistista toiseen muistiin.
DP	DisplayPort – Videoliitäntästandardi.
DVI	Digital Video Interface – Videoliitäntästandardi.
FIFO	First In, First Out – Tapa järjestellä datapuskuri. Ensin sisään tullut data otetaan myös ensimmäisenä ulos.
FPGA	Field Programmable Gate Array – Eräs ohjelmoitavan logiikan tyyppi.
FW	Firmware – Laitteen sisällä oleva kiinteästi laitteeseen liittyvä ohjelmisto. UFG:ssä firmware sisältää myös FPGA:n konfiguraation.
HDMI	High Definition Media Interface – Videoliitäntästandardi.
HPD	Hot Plug Detect – Eräiden multimedialiitäntöjen (esim. DP, HDMI) menetelmä havaita kaapelin irrotus- ja kiinnitystapahtumat.
HW	Hardware – Laitteisto.
HWIO	Hardware I/O – Suoraan laitteiston kanssa tapahtuva luku ja kirjoitus.
I/O	Input/Output – Sisään-/ulostulo.
LVDS	Low Voltage Differential Signalling – Videoliitäntästandardi.
OLVDS	Octal LVDS – 8-porttinen LVDS.

PCI	Peripheral Component Interface – Rinnakkaismuotoinen tietokoneen oheislaiteväylästandardi.
PCIe	PCI Express – Sarjamuotoinen tietokoneen oheislaiteväylästandardi.
QLVDS	Quad LVDS – 4-porttinen LVDS.
SDK	Software Development Kit – Ohjelmistokehityspaketti, joka liittyy johonkin ohjelmointirajapintaan tai laajempaan ohjelmointialustaan. Paketti sisältää tavallisesti ohjeita ja esimerkkiohjelmia, joiden avulla ohjelmoija voi tehdä omia sovelluksia kyseiselle alustalle.
SDRAM	Synchronous Dynamic Random Access Memory – Eräs dynaamisen muistin tyyppi.
SW	Software – Ohjelmisto.
TCL	Test Control Language – Testausjärjestelmissä käytetty ajonaikaisesti tulkattava ohjelmointikieli.
UFG	Unigraf Frame Grabber – Unigraf oy:n videokaappauslaite
UUF	Unigraf Update File – Unigraf oy:n oma ohjelmistopäivitystiedostomuoto.
VCL	Visual Component Library, C++ Builder -työkalun mukana tuleva komponenttikirjasto.

SISÄLLYS

1 JOHDANTO

Unigraf oy on videotestauslaitteistoja ja -ohjelmistoja kehittävä, valmistava ja markkinoiva yritys. Tuotteisiin kuuluu videotestigeneraattoreita, kuvankaappauslaitteita, analysointityökaluja ja määräystenmukaisuustestauksen (compliance test) työkaluja. Tuotteet ovat olleet Unigrafin historian alusta asti uraauurtavia tiennäyttäjiä, mikä on tarjonnut etulyöntiaseman suppeilla ja voimakkaasti erikoistuneilla videomittalaitemarkkinoilla.

Unigraf jatkaa koko ajan askel edellä kilpailijoitaan ja on osallistunut alan standardien ja käytäntöjen luomiseen yhteistyössä maailman suurimpien standardointijärjestöjen ja elektroniikkavalmistajien kanssa. Tuotteet ovat tunnettuja maailmanlaajuisesti erityisesti päämarkkina-alueilla Aasiassa ja Pohjois-Amerikassa. Niitä käytetään tuotekehityslaboratorioissa ja tehtaiden tuotantolinjoilla sekä jonkin verran laivoissa ja lentokoneissa. Unigraf on valmistanut myös teollisuustietokoneita, näyttöjä, projektorinäyttötauluja ja junatietokoneiden käyttöliittymiä.

Unigraf oy:n historia alkaa vuodesta 1971 Kone Instrumentsin tuotekehitysosastosta, josta näyttötekniikkaan ja teollisuuden pariin siirtyminen alkoi 1970-luvun loppupuolella. Omaksi yrityksekseen Unigraf irrottautui vuonna 1990, jolloin osaston johto osti liiketoiminnan itselleen. Yrityksen henkilömäärä on noin 20 ja toimipiste sijaitsee Espoossa. Tuotteet kootaan ja testataan Espoon toimipisteessä. Unigraf käyttää lisäksi alihankintaa niin tuotekehityksessä kuin tuotannossakin.

Tuotteissa pyritään laatuun, mukautuvuuteen, toimintavarmuuteen ja helppokäyttöisyyteen. Tämän vuoksi tuotekehitystä tehdään jatkuvasti. Opinnäytetyössä toteutettavalla laiteohjauskirjastolla on tärkeä rooli osana tuotestrategiaa. Se on sovelluslaajennus, jota asiakkaiden tekemät ja Unigrafin omat sovellukset voivat käyttää UFG-04- ja UFG-06-tuotteiden ohjaukseen.

Aikaisempi ratkaisu, jossa jokaiselle laitteelle on ollut oma yksilöllinen ohjauskirjasto ja näille yhteinen ”kattokirjasto”, on todettu monimutkaiseksi ylläpitää ja dokumentoida. Lisäksi asiakkaat ovat kokeneet sen hankalaksi käyttää. Aikaisempi

ratkaisu ei ole tukenut moniajoa lainkaan. Opinnäytetyössä toteutettava uusi järjestelmä tulee korvaamaan aikaisemman vaiheittain.

Uuden kirjaston toteutuksessa on tärkeää ottaa huomioon, missä tilanteissa sitä voidaan käyttää, mitä laitteita sen tulee tukea ja mitä toimintoja sen tulee sisältää. Opinnäytetyön tavoitteena on ensisijaisesti selvittää, miten UFG-laiteohjauskirjaston toteutuksessa voidaan ottaa huomioon mahdollisuus usean laitteen yhtäaikaiseen käyttöön useasta eri sovelluksesta ja miten sovelluksen säikeistys vaikuttaa toteutukseen. Opinnäytetyö pyrkii vastaamaan kysymykseen, miten oliopohjaisella toteutustavalla ja Windows-käyttöjärjestelmän tarjoamilla suojausmenetelmillä voidaan suojata laiteresursseja ja ohjelman tiloja moniajon aiheuttamilta ongelmilta DLL-kirjastossa, jossa funktioiden kutsujärjestys ei ole etukäteen tunnettu. Lisäksi sivutaan sitä, miten ohjelmointirajapinnassa voidaan ottaa huomioon tuki erilaisille ohjelmointikielille.

Aluksi opinnäytetyössä käsitellään työn taustoja ja rajauksia. Seuraavaksi esitellään moniajoa nykyaikaisissa käyttöjärjestelmissä, muutamia yleisiä suojausmenetelmiä ja niistä erityisesti mutexia. Kolmanneksi käydään läpi laiteohjauskirjaston suunnittelua, toteutusta ja teorian soveltamista käytännössä. Viimeiseksi tarkastellaan opinnäytetyön onnistumista ja jatkokehitysmahdollisuuksia.

2 TYÖN TAUSTAT JA RAJAUS

2.1 Kehittämistarve ja tavoitteet

Tarve uudelle laiteohjauskirjastolle syntyi teknologian kehittymisen myötä ja asiakasvaatimusten muuttuessa. Asiakkaat olivat aikaisemmin käyttäneet yksinkertaisia testijärjestelmiä, jotka suorittivat ennalta määrätyn sekvenssin yhden kerran alusta loppuun. Näiden rinnalle olivat tulossa jatkuva-aikaiset testijärjestelmät, joissa hyödynnettiin moniajoa.

Unigrafin uusin videotestigeneraattori oli saatu juuri valmiiksi markkinoita varten. Projektista vapautui henkilöresursseja, joten uuden UFG-laiteohjauskirjaston kehittäminen päätettiin aloittaa osana UFG-04-tuoteperheen tuotteistamisprosessia.

Nykyiset UFG-laiteohjauskirjastot oli teetetty suurimmaksi osaksi alihankintana, eikä niiden tekninen toteutus sallinut joustavasti muutosten tekemistä uusiin haasteisiin vastaamiseksi. Tämän lisäksi epäjatkuvuudet versioiden välisessä yhteensopivuudessa, puutteellinen moniajotuki ja ohjelmointirajapinnan huono yhteensopivuus eri ohjelmointikielten kanssa vaikuttivat tarpeeseen kehittää kokonaan uusi laiteohjauskirjasto. Uudesta kirjastosta haluttiin yksinkertainen, joustava ja soveltuva sekä nykyisille että tuleville UFG-laitetyypeille.

Yhteensopivuus eri versioiden välillä on erityisen tärkeää asiakkaiden kannalta. Jos ohjelmointirajapinta muuttuu lähes joka kerta, kun uusi versio julkaistaan, asiakkaat joutuvat uudelleen ja uudelleen muuttamaan omia ohjelmiaan ottaakseen rajapinnan käyttöön. Jos asiakas valmistaa tuotteita myytäväksi edelleen omille asiakkailleen, voi uuden version käyttöönotto loppuasiakkaan kohdalla olla jopa mahdotonta. Esimerkiksi tuotantolinjoilla päivitys tarkoittaa yleensä tuotantoseisokkia, mikä voi tulla kalliiksi loppuasiakkaalle.

Yhteensopivuustakuu, käyttöturvallisuus ja monipuolinen ohjelmointikielten tuki tuottavat huomattavasti lisäarvoa UFG-tuotteille asiakkaiden näkökulmasta, ja sisäinen joustavuus ja yleiskäyttöisyys lyhentävät oleellisesti tuotekehitysaikaa Unigrafin omissa tuotekehitysprojekteissa. Projekti on näin ollen perusteltu ja tervetullut uudistus Unigrafin tuotestrategiaan.

2.2 Rajaukset

Tässä opinnäytetyössä käsitellään laiteohjauskirjaston suunnittelua ja toteutusta olemassa oleville laitteille. Työssä viitataan UFG-laitteisiin, kuvankaappaustekniikkaan ja erilaisiin video- ja multimediastandardeihin, mutta näiden toteutukseen ja teknisiin yksityiskohtiin ei paneuduta enempää kuin työn ymmärtämisen kannalta on välttämätöntä.

Opinnäytetyössä viitattuja olemassa olevia laiteohjauskirjastoja ja niiden rajapintoja ei esitellä. Työssä käydään läpi vain vanhojen kirjastojen toteutustapaa ja yleisiä ominaisuuksia sekä niiden vaikutusta uuden kirjaston kehitystyöhön.

Moniajoa esitellään sen verran, että lukija ymmärtää moniajon käsitteenä ja pystyy hahmottamaan, miten moniajo vaikuttaa ohjelmiston rakenteellisiin vaatimuksiin. Käyttöjärjestelmien ja moniprosessorijärjestelmien erilaisia moniajototeutuksia käsitellään vain sen verran, että ongelmakentästä saadaan kohtuullinen yleiskuva.

Työssä esitellään erilaisia käyttöjärjestelmien tarjoamia suojausmenetelmiä, suojaukseen liittyviä ongelmia ja ohjelmarakenteellisia ratkaisuja. Työn toteutuksessa käytetään esiteltävistä suojausmenetelmistä ainoastaan mutexia, joten siitä käydään läpi sekä toimintaperiaate että käyttö Windowsissa. Muiden esiteltävien suojausmenetelmien toimintaperiaatetta tutkitaan vertailun vuoksi, mutta niiden käyttöön Windowsissa ei perehdytä.

3 MONIAJO

3.1 Tausta ja periaate

Moniajo on ollut mahdollista tietokoneissa jo 1960-luvulta lähtien, jolloin IBM julkisti vuonna 1964 ensimmäisen moniajoa tukevan tietokonejärjestelmäperheen, System/360:n tai lyhemmin S/360 (IBM 2011). Pohjalla oli Gene Amdahlin (s. 16.10.1922) pitkäjänteinen kehitystyö, jota sivuttiin jo hänen tohtorin väitöskirjassaan (Amdahl 1951). Laitteistolla onkin yhtä tärkeä osuus moniajon kannalta kuin ohjelmistolla (Olsson & Piani 1992, 195).

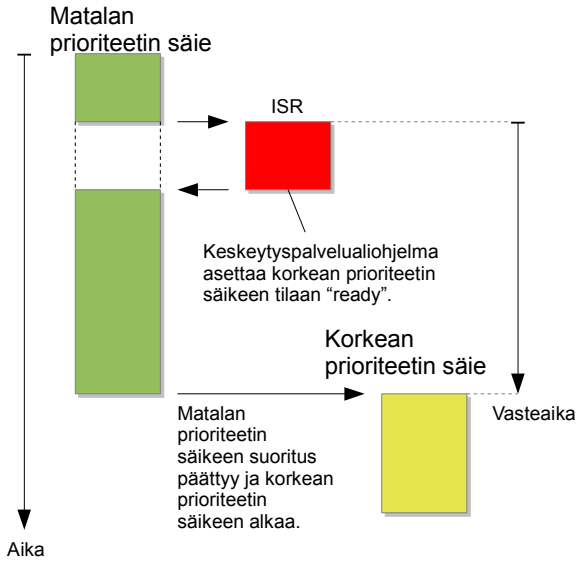
Moniajossa eri tehtäviä voidaan suorittaa samanaikaisesti. Se lisää prosessorin käyttöastetta ja näin ollen tehostaa sovelluksen toimintaa (Labrosse 2002, 37). Moniajolla voidaan tarkoittaa eri tietokoneiden välillä tapahtuvaa yhtäaikaista tehtävien suorittamista, prosessorien välistä tehtävien jakoa, prosessoriytimien välillä tapahtuvaa vuorottelua tai yhden prosessorin sisällön vuorottelua. Moniajossa tehtävät käyttävät useimmiten samoja resursseja, mutta toimivat muuten lähes kokonaan toisistaan riippumatta. (Olsson & Piani 1992, 198.)

3.2 Keskeyttävä ja ei-keskeyttävä moniajo

Moniajojärjestelmää, jossa jokainen säie suorittaa tehtävän loppuun asti ennen kuin antaa vuoron toiselle säikeelle, kutsutaan ei-keskeyttäväksi moniajoksi (non-preemptive multitasking tai cooperative multitasking). Tällaisessa järjestelmässä säie suoritetaan loppuun, ennen kuin seuraavan säikeen suoritus alkaa. Säikeet voivat myös sopia vuorottelusta keskenään. (Labrosse 2002, 40.)

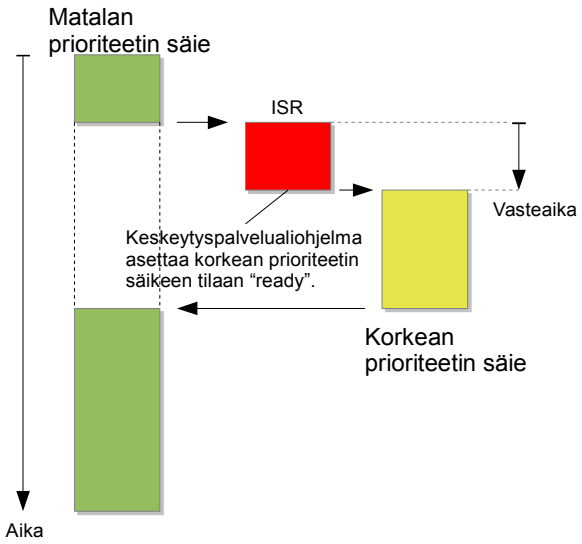
Ei-keskeyttävässä moniajossa (kuvio 1) satunnainen tapahtuma (asynchronous event) käynnistetään keskeytyspalvelualiohjelmassa (ISR), joka asettaa tapahtuman suorittavan korkeaprioriteettisen säikeen valmiustilaan. Tämän jälkeen pala-

taan takaisin suoritettavana olevaan säikeeseen. Tapahtuman käsittely alkaa vasta kun suoritettavana oleva säie saa tehtävän valmiiksi. Satunnaistapahtuman vasteaika vaihtelee riippuen siitä, missä vaiheessa pyyntö tapahtuu suhteessa suoritettavana olevaan säikeeseen. (Labrosse 2002, 40 - 41.)



KUVIO 1. Ei-keskeyttävän moniajon periaate (Labrosse 2002, 41)

Vaihtoehto ei-keskeyttävälle moniajolle on keskeyttävä eli pre-emptiivinen moniajo (preemptive multitasking) (kuvio 2), jossa suoritettavana on aina korkeimman prioriteetin säie. Keskeytyspalvelualiohjelma asettaa satunnaistapahtuman suorittavan korkeaprioriteettisen säikeen valmiustilaan, jolloin vuorottelija (scheduler) ottaa sen välittömästi suoritettavaksi. Kun tehtävä valmistuu, palataan suorittamaan edellistä säiettä. Jos korkeimman prioriteetin säikeitä on samanaikaisesti useita, suoritusaika voidaan jakaa tasaisesti näiden välillä (time slicing). Tällaista vuorottelua kutsutaan nimellä Round-Robin. (Labrosse 2002, 43 - 45.)



KUVIO 2. Keskeyttävän moniajon periaate (Labrosse 2002, 42)

Nykyisten Windows-käyttöjärjestelmän versioiden ydin (kernel) perustuu NT-teknologiaan, joka on pre-emptiivinen. Tämän vuoksi moniajoa ja sen problematiikkaa rajoitutaan tarkastelemaan vain pre-emptiivisen moniajon kannalta.

3.3 Prosessit ja säikeet

Käyttöjärjestelmä on monimutkainen ohjelmistokokonaisuus, joka tarjoaa prosesseille virtuaalisen ympäristön resursseineen ja suoritinaikoineen (Olsson & Piani 1992, 199).

> *Käyttöjärjestelmä on vastuussa prosessien hallinnasta ja niiden välisestä tiedonsiirrosta ja suojauksesta. Käyttöjärjestelmä luo prosessin jokaiselle suoritettavalle ohjelmalle. Lisäksi prosessit voivat luoda toisia prosesseja, joita kutsutaan lapsiprosesseiksi. Prosessit voivat olla näennäisesti yhtäaikaisia, eli useita prosesseja voi olla suoritettavana samanaikaisesti. Käyttöjärjestelmä jakaa prosessoriaikaa näiden prosessien välillä niiden prioriteetista ja tilasta riippuen. (Silberschatz, Peterson & Galvin 1991, 87 - 93. Kirjoittajan suomennos.)*

Käyttöjärjestelmä varaa jokaiselle prosessille tarvittavat resurssit. Esimerkiksi Windows-käyttöjärjestelmässä jokaiselle prosessille varataan virtuaalinen muisti-

alue, jonka koko on 32-bittisessä järjestelmässä neljä gigatavua. Siitä kaksi gigatavua on varattu suoritettavalle ohjelmakoodille ja Windows-käyttöjärjestelmän prosessikohtaisille objekteille, ja toiset kaksi gigatavua ovat prosessin itsensä tekemien muistivarausten käytettävissä. (MSDN 2011c.)

Pienten rinnakkaisten tehtävien suorittamiseen prosessi on melko raskas väline, ja siksi käyttöjärjestelmät tarjoavat kevyempiäkin menetelmiä. Yksinkertaisin menetelmä on haarauttaminen, jossa yhdellä funktiokutsulla luodaan ”lapsiprosessi”, joka on lähes täydellinen kopio isäntäprosessista. Windowsissa haarauttaminen voidaan tehdä spawn-funktiolla (MSDN 2011j) ja Unix-pohjaisissa järjestelmissä, kuten Linuxissa, fork-funktiolla (LinuxManPages 1995). Linuxin käyttöoppaan mukaan ainoat erot haarauttamalla luodun lapsiprosessin ja isäntäprosessin välillä ovat prosessitunniste (PID, Process Identifier), isäntäprosessin tunniste (PPID, Parent Process Identifier) ja järjestelmäresurssien käyttö. Windowsissa joidenkin resurssien, esimerkiksi avoimien tiedostojen, periytymisen lapsiprosessille voi sen sijaan määrätä spawn-funktiokutsun yhteydessä (MSDN 2011j).

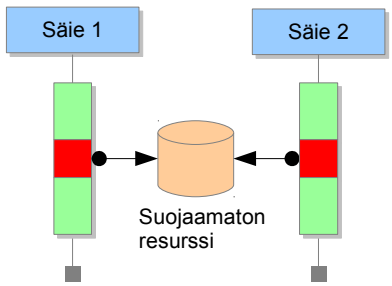
Sekä Windowsissa että Linuxissa voidaan käyttää säikeitä, jotka ovat haarauttamisen tavoin prosessia kevyempi vaihtoehto. Todellisuudessa prosessi sisältää aina vähintään yhden säikeen (pääsäie), joka luodaan prosessin yhteydessä. Kaikki prosessin pääohjelmatason toiminnot suoritetaan pääsäikeessä. Prosessin ja säikeen merkittävin ero on se, että säikeelle ei varata kaikkia erillisiä resursseja prosessin tavoin, vaan se käyttää osaksi isäntäprosessin resursseja. Säikeelle varataan vain sen toiminnan kannalta välttämättömät käyttöjärjestelmäresurssit, esimerkiksi pino ja muistia eräitä prosessorin rekistereitä varten. (MSDN 2011g.)

3.4 Jaetut resurssit

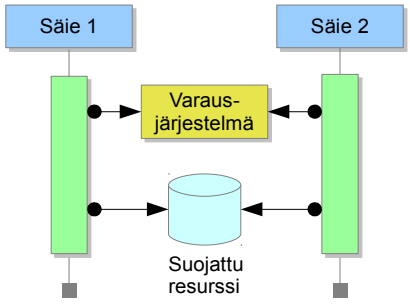
Järjestelmän resursseja, joita voi käyttää useampi kuin yksi säie kerrallaan, ovat muun muassa muistit, säikeiden välinen tiedonsiirto ja oheislaitteet. Näitä resursseja kutsutaan joissakin lähteissä jaetuiksi resursseiksi (shared resource) tai toisis-

sa suojatuiksi resursseiksi (protected resource). Useimmiten näillä kuitenkin tarkoitetaan samaa asiaa. Ohjelmakoodin aluetta, jossa jaettua resurssia käytetään, kutsutaan kriittiseksi alueeksi (critical section tai critical region).

Jos samaa resurssia käyttää useampi kuin yksi säie yhtä aikaa, säikeiden välillä voi syntyä ristiriita (kuvio 3), joka voi johtaa tiedon vahingoittumiseen ja odottamattomiin ongelmiin, jopa järjestelmän täydelliseen kaatumiseen. Jotta ristiriidoilta vältyttäisiin, jaettu resurssi täytyy suojata eli sen päällekkäinen käyttö estää. Tämä voidaan toteuttaa luomalla sille varausjärjestelmä (kuvio 4). Ennen kuin säie tulee kriittiselle alueelle, se odottaa alueella käytettävän resurssin vapautumista. Kun resurssi vapautuu, tai jos se on jo valmiiksi vapaa, säie tulee kriittiselle alueelle ja varaa samalla resurssin itselleen. Poistuessaan kriittiseltä alueelta säie vapauttaa resurssin. (Olsson & Piani 1992, 206 - 207.)



KUVIO 3. Resurssiristiriita



KUVIO 4. Resurssin suojaus varausjärjestelmällä

4 SUOJAUS

4.1 Kriittinen alue

Kriittisellä alueella on enemmän käsitteellinen kuin käytännöllinen merkitys ohjelmoinnissa, koska sitä ei ole toteutettu näkyvässä muodossa missään ohjelmointikielessä (Olsson & Piani 1992, 211). Ohjelmakoodin kriittisellä alueella tarkoitetaan kohtaa, jossa käsitellään jaettua resurssia.

Monissa yksiprosessoristen järjestelmien reaaliaikakäyttöjärjestelmissä kriittisen alueen suojaus on toteutettu kieltämällä keskeytykset ja samalla pysäyttämällä säikeiden vuorottelija varauksen ajaksi. Tämä tarkoittaa sitä, että kaikkien muiden säikeiden suoritus keskeytyy kokonaan niin pitkäksi aikaa kuin toiminta kriittisellä alueella kestää. (Olsson & Piani 1992, 211; Silberschatz ym. 1991, 139.) Esimerkiksi MicroC/OS-II:ssa on tätä varten funktio OS_ENTER_CRITICAL() kriittiselle alueelle siirtymistä ja funktio OS_EXIT_CRITICAL() sieltä poistumista varten (Labrosse 2002, 74).

Ongelmana keskeytysten kieltämisessä on se, että siinä hukataan olennainen osa yhtäaikaisen prosessoinnin tehokkuudesta varaamalla prosessorin yhdelle säikeelle kerrallaan. Jos kriittisiä alueita on koodissa paljon tai ne vievät pitkiä suoritusaikoja, voi koko järjestelmän suorituskyky kärsiä. (Silberschatz ym.1991, 139.) Lisäksi keskeytysten kieltämistä ei voida käyttää suojauskeinona moniprosessorijärjestelmissä.

Kriittisen alueen käsitettä ei tule sekoittaa Windowsin CriticalSection-nimiseen suojausobjektiin. Kyseinen objekti on mutex-tyyppinen ja prosessikohtainen. Se ei kiellä keskeytyksiä eikä estä toisten säikeiden samanaikaista suorittamista, ja sitä voidaan käyttää sekä yksi- että moniprosessorijärjestelmissä. (MSDN 2011d.)

4.2 Vapaakäyntinen ohjelmalohko

Ohjelmalohkoa, joka pysyy koko ajan muuttumattomana, kutsutaan vapaakäyntiseksi (reentrant). Muuttumattomuus tarkoittaa sitä, että lohkon tila ei muutu eri suorituskertojen välillä. Vapaakäyntistä lohkoa voidaan suorittaa eri säikeistä yhtä aikaa tarvitsematta pelätä sen aiheuttavan resurssiristiriitoja. (Silberschatz ym. 1991, 257.)

Vapaakäyntinen ohjelmalohko on proseduraalisissa ohjelmointikielissä käytännössä sama asia kuin vapaakäyntinen funktio. Tällainen funktio ei käytä globaaleja muuttujia, vaan kaikki sen käsittelemät arvot annetaan osoitin- tai arvoparametreina, jolloin ne talletetaan pinoon. (Labrosse 2002, 43 - 44.) Kuten luvussa Prosessit ja säikeet sivulla 8 todettiin, pino on säiekohtainen. Funktio käsittelee tällöin ainoastaan säikeen sisäisiä arvoja, joten sitä voidaan ajaa yhtä aikaa useissa säikeissä. Tästä voidaan tehdä myös se johtopäätös, että luokan jäsenfunktio on vapaakäyntinen, jos se ei käsittele luokan jäsenmuuttujia eikä globaaleja muuttujia.

Jos lohko ei ole luonnostaan vapaakäyntinen, eli se sisältää kriittisen alueen, siitä voidaan tehdä vapaakäyntinen varaamalla ja vapauttamalla kriittinen alue lohkon sisällä. Pre-emptiivisessä moniajossa kaikkien samanaikaisesti suoritettavien lohkojen on oltava vapaakäyntisiä. (Labrosse 2002, 43.)

4.3 Spinlock

Spinlockin eli pyörityslukon avulla voidaan toteuttaa resurssien suojaus ilman keskeytysten kieltämistä ja muiden säikeiden suorituksen pysäyttämistä. Spinlock perustuu boolean-tyyppiseen muuttujaan (lock) ja prosessorin Test-and-Set-ohjelmakäskyyn. Test-and-Set-käsky tarkistaa ja asettaa muuttujan sisällön yhden atomisen operaation aikana, jolloin mikään muu säie ei voi lukea tai asettaa sitä samanaikaisesti. Moniydin- ja moniprosessorijärjestelmissä Test-and-Set-käskyn samanaikaisuus eri prosessoreissa on estetty fyysisesti prosessorien välisessä siltaukses-

sa, joten myös niissä käskyt suoritetaan aina peräkkäin jossain ennalta määrätyssä järjestyksessä. (Silberschatz ym. 1991, 147.)

Spinlockilla on kaksi tilaa, true (locked) ja false (unlocked). Suojattava resurssi on lukittu, kun tila on true, ja vapautettu, kun tila on false. Säie odottaa lukituksen vapautumista tarkistamalla lukon tilaa jatkuvasti ”pollaamalla” eli ”spinnaamalla”. Jatkuva tilan tarkistaminen kuitenkin tuhlaa prosessoriaikaa säikeen tehdessä turhia Test-and-Set-kutsuja. Tuhlaus voidaan estää käyttämällä spinlockin sijasta odotusjonoa. (Silberschatz ym. 1991, 152.)

Spinlock on moniprosessorijärjestelmässä yksi tapa estää eri prosessoreissa suoritettavia säikeitä käyttämästä samaa jaettua resurssia yhtä aikaa (Oney 2002, 172 - 173). Windowsissa spinlockia käytetään sellaisenaan vain kernel-tasolla, eli käyttöjärjestelmän sisäisessä toteutuksessa ja ajureissa (MSDN 2011e).

Nykyisin spinlockia käytetään myös ehkäisemään säikeen tarvetta siirtyä odotusjonoon muiden suojausprimitiivien, esimerkiksi mutexin, yhteydessä. Jos resurssi varataan lähes aina lyhyeksi aikaa kerrallaan, säikeen siirtyminen odotusjonoon ja sieltä poistuminen voi olla hidasta suhteessa varauksen kestoon. Tällöin on perusteltua käyttää ensin spinlockia tutkimaan varauksen tilaa odotettavissa olevan vapautumisajan verran, ja vasta odotusajan pitkittyessä siirtyä odotusjonoon. Esimerkkinä spinlockia käyttävästä mutex-tyyppisestä suojausobjektista on Windows-käyttöjärjestelmän CriticalSection-objekti (MSDN 2011d).

4.4 Semafori

Semaforin sanotaan olevan yleisin suojausmenetelmistä. Sen kehitti hollantilainen tiedemies Edsger Dijkstra vuonna 1968 korvaamaan aiemmin käytetyt erilliset suojausmuuttujat (Olsson & Piani 1992, 213). Ensimmäinen semafori oli binäärinen (binary semaphore), ja sen idean pohjalta toinen hollantilainen tiedemies, tohtori Carel S. Scholten, kehitti myöhemmin laskevan semaforin (counting semap-

hore), josta käytetään myös nimitystä yleinen semafori (general semaphore). (Cooling 2011s.)

Määritelmän mukaan semafori on kokonaislukumuuttuja S, joka voi saada arvot $S \geq 0$. Siihen liittyy kaksi atomista operaatiota, odotus (wait) ja merkinanto (signal) (Silberschatz ym. 1991, 150). Dijkstra antoi operaatioille nimet P ja V, jotka Silberschatzin ym. mukaan tulevat hollannin kielen sanoista proberen (testata) ja verhogen (suurentaa). Coolingin mukaan sanan proberen sijasta P:llä tarkoitettiin uudissanaa prolagen (yrittää ja pienentää), mikä sopiikin paremmin kuvaamaan semaforin toimintaa.

Semafori voidaan varata, jos sen arvo on suurempi kuin nolla (kuvio 5). Varaaminen vähentää semaforin arvoa yhdellä, ja vapautus taas vastaavasti kasvattaa sitä. Tästä voidaan päätellä, että semaforille on annettava luotaessa alkuarvo $S > 0$, joka kertoo varattavissa olevien resurssien määrän.

```
P(S): while S ≤ 0 do NOP
      S = S - 1

V(S): S = S + 1
```

KUVIO 5. Klassinen määritelmä semaforin wait- ja signal-operaatioille

Jos semaforia odotetaan silmukassa, se tuhlaa prosessoriaikaa samalla tavalla kuin spinlock. Tämä voidaan estää siten, että semaforin toteutuksessa luodaan sille odotusjono, johon säie siirtyy wait-operaation yhteydessä. Samalla säie pysäyttää toimintansa jonossa olon ajaksi. Kun semafori vapautetaan, signal-operaatio käynnistää automaattisesti seuraavana jonossa olevan säikeen, jolloin tämä voi aloittaa jaetun resurssin käytön. (Silberschatz ym. 1991, 152.)

4.5 Mutex

Nimi mutex tulee sanoista mutual exclusion eli keskinäinen poissulkeminen. Itse asiassa kaikki opinnäytetyössä esitellyt suojausmenetelmät perustuvat samaan keskinäisen poissulkemisen ajatukseen.

Mutex perustuu suojausmenetelmänä semaforiin. Binäärisen semaforin ja toisaalta alkuarvon 1 laskevan semaforin onkin väitetty olevan sama asia kuin mutex (Silberschatz ym. 1991, 151). Cooling kuitenkin varoittaa blogikirjoituksessaan sekoittamasta näitä keskenään, koska hänen mukaansa se johtaa vakaviin ongelmiin. Näitä ovat esimerkiksi äkillinen vapauttaminen, rekursiivinen kuollut lukko (deadlock) tai tehtävän odottamattomasta päättymisestä (task death) johtuva kuollut lukko. (Cooling 2011a.)

Coolingin mukaan merkittävin ero mutexin ja semaforin välillä on omistajuusperiaate; mutexin omistaa se säie, joka sen on luonut. Vain kyseinen säie voi vapauttaa mutexin. Sen sijaan semaforilla ei ole omistajaa, joten mikä tahansa säie voi vapauttaa sen. Semaforin käyttöä mutexin tilalla tulisikin välttää. (Cooling 2011b.)

Semaforilla ja mutexilla voidaan todeta toinenkin perustavanlaatuinen ero, joka liittyy niiden käyttäytymiseen saman säikeen varatessa niitä uudelleen. Semaforin laskuri muuttuu joka kerta säikeen varatessa sen. Jos laskuri saavuttaa nollan, säie jumiutuu, kunnes jokin muu säie vapauttaa ainakin yhden varauksen. Jos säie yrittää varata semaforin useammin kuin sen alkuarvo, säie jää kuolleeseen lukkoon. Mutexia sen sijaan voi varata lähes rajattomasti samasta säikeestä ilman, että säie jumiutuu.

4.6 Kuollut lukko

Kuollut lukko (deadlock) on tila, jossa yksi tai useampi säie odottaa tapahtumaa, jota ei koskaan tule. Jos odotettavan tapahtuman käynnistävä säie on itsekin odot-

tavassa tilassa, kaikki säikeet odottavat loputtomasti eli ovat kuolleessa lukossa. Toinen vaihtoehto on se, että jokin säikeistä toimii, mutta ei voi selvittää kuolleen lukon tilannetta. Tällöin puhutaan ”nälkiintymisestä” (starvation). (Olsson & Piani 1992, 211.)

Kuolleeseen lukkoon voidaan joutua järjestelmässä, jossa kaikki neljä seuraavista ehdoista täyttyvät:

- Järjestelmässä käytetään keskinäistä poissulkemista (mutual exclusion).
- Resurssin voi vapauttaa vain säie, joka varasi sen (non-preempted allocation).
- Säie voi varata resursseja yksi kerrallaan (successive allocation).
- Säikeet voivat varata resursseja toisistaan riippumattomassa järjestyksessä (inverse-order allocation). (Olsson & Piani 1992, 212.)

Näistä minkä tahansa puuttuminen estää kuolleen lukon syntymisen, mistä voidaan johtaa seuraavat neljä vaihtoehtoista ratkaisutapaa:

- Järjestelmässä ei käytetä keskinäistä poissulkemista.
- Käyttöjärjestelmä vapauttaa resurssin väkisin havaitessaan kuolleen lukon.
- Säie varaa aina kaikki resurssit kerralla tarvitessaan yhtä resurssia.
- Säikeet varaavat resurssit aina ennalta määrätyssä järjestyksessä. (Olsson & Piani 1992, 212 - 213.)

Voidaan todeta, että ainoa yllä mainituista ratkaisutavoista, jota voidaan soveltaa käytännössä, on etukäteen sovittu varausjärjestys. Tätä tapaa on helppo soveltaa, kun kaikki säikeet toteuttaa sama ohjelmoija. Tilanne muuttuu vaikeammaksi, jos säikeiden ohjelmoijat eivät voi saada etukäteen selville varausjärjestystä. (Olsson & Piani 1992, 212 - 213.)

5 WINDOWS-KÄYTTÖJÄRJESTELMÄN MUTEX-OBJEKTI

5.1 Luonti ja tuhoaminen

Windows-käyttöjärjestelmässä mutex-objekti luodaan CreateMutex-funktiolla, jonka prototyyppi on esitelty kuviossa 6. Samalla funktiolla voidaan luoda sekä nimetty että nimeämätön objekti. Jos luominen onnistuu, funktio palauttaa ”kahvan” (handle), jonka avulla objektia voidaan käyttää muista funktioista. Jos luominen sen sijaan epäonnistuu, funktio palauttaa arvon NULL. Tällöin epäonnistumiseen johtaneen virheen koodi voidaan hakea GetLastError-funktiolla. (MSDN 2011b.)

```
1   HANDLE WINAPI CreateMutex(
2       __in_opt LPSECURITY_ATTRIBUTES lpMutexAttributes,
3       __in     BOOL bInitialOwner,
4       __in_opt LPCTSTR lpName
5   );
```

KUVIO 6. CreateMutex-funktion prototyyppi

Kaikki Windows-käyttöjärjestelmän objektit, joihin on olemassa kahva, voidaan sulkea CloseHandle-funktiolla. Tämä pätee myös mutex-objektiin. (MSDN 2011a.)

5.2 Turvallisuusattribuutit

Turvallisuusattribuuteilla voidaan määrittää, millä prosessilla on oikeus avata nimetty mutex-objekti. Jos lpMutexAttributes-parametrille (kuvio 6, rivi 2) annetaan arvo NULL, objektin kahvaa ei voi jakaa lapsiprosesseille (MSDN 2011b).

Turvallisuusattribuutteja käytetään vain, jos objektille on välttämätöntä määritellä sen omistavan isäntäprosessin attribuuteista poikkeava turvallisuuskäytäntö. Muussa tapauksessa objekti perii oletuksena isäntäprosessin turvallisuusattribuutit. (MSDN 2011b.)

5.3 Nimetty mutex

Mutex-objektille on mahdollista antaa luotaessa nimi. Nimi tulee antaa, jos objekti on tarkoitus jakaa prosessien välillä. Nimeämätöntä mutex-objektia voidaan käyttää vain prosessin sisäisten säikeiden välillä. (MSDN 2011b.)

Nimi annetaan objektia luotaessa CreateMutex-funktion lpName-parametrilla (kuvio 6 sivulla 16, rivi 4). Jos annetun niminen objekti on jo olemassa ja prosessilla on riittävät oikeudet avata kyseinen objekti, funktio palauttaa kahvan olemassa olevaan objektiin. Tällöin GetLastError-funktio palauttaa koodin ERROR_ALREADY_EXISTS. (MSDN 2011b.)

Jo olemassa oleva nimetty mutex-objekti voidaan avata joko CreateMutex- tai kuviossa 7 esitellyllä OpenMutex-funktiolla. Näiden erona on, että OpenMutex-funktio palauttaa aina virheen, jos annetulla nimellä varustettua objektia ei ole olemassa, kun taas CreateMutex-funktio luo vastaavassa tilanteessa uuden objektin. (MSDN 2011b.)

```
1   HANDLE WINAPI OpenMutex(
2       __in DWORD dwDesiredAccess,
3       __in BOOL bInheritHandle,
4       __in LPCTSTR lpName
5   );
```

KUVIO 7. OpenMutex-funktion prototyyppi

5.4 Alkuperäinen omistaja

Mutex-objektille voidaan määritellä alkuperäinen omistaja (initial owner). Kun säie luo mutex-objektin, se kertoo bInitialOwner-parametrilla (kuvio 6 sivulla 16, rivi 3), haluaako se olla objektin alkuperäinen omistaja. Jos parametrin arvo on false, mikä tahansa muu säie voi asettua mutex-objektin alkuperäiseksi omistajaksi avatessaan objektin CreateMutex-funktiolla. (MSDN 2011b.)

Mikäli mutex-objektin luoja vaatii alkuperäistä omistajuutta itselleen ja jokin toinen säie omistaa sen jo, CreateMutex-funktio palauttaa virheen ERROR_ALREADY_EXISTS. Mutex-objektin alkuperäinen omistaja on vastuussa objektin vapauttamisesta. (MSDN 2011b.)

5.5 Varaaminen ja vapauttaminen

Kuten muitakin suojausmenetelmiä, säikeen täytyy odottaa mutex-objektia ennen siirtymistä sen suojaamalle kriittiselle alueelle. Objekti varataan automaattisesti odotuksen päättyessä onnistuneesti, eli säikeen päästessä kriittiselle alueelle. Kun säie poistuu kriittiseltä alueelta, sen tulee vapauttaa varaamansa objekti ReleaseMutex-funktiolla. (MSDN 2011f.) Funktion prototyyppi on esitetty kuviossa 8.

```
1   HANDLE WINAPI ReleaseMutex(
2       __in HANDLE hMutex
3   );
```

KUVIO 8. ReleaseMutex-funktion prototyyppi

5.6 Odottaminen

Windows-käyttöjärjestelmässä mutex-objektia odotetaan WaitForSingleObject-funktiolla, joka on esitelty kuviossa 9. Odotusaika voidaan määritellä parametrin dwMilliseconds (kuvio 9, rivi 3) arvolla. Odotus voidaan määritellä myös päättymättömäksi antamalla ajan sijasta vakio INFINITE, tai välittömästi palaavaksi antamalla arvoksi nolla. Jälkimmäisessä tapauksessa funktio palaa heti riippumatta siitä, saatiinko objektin varattua vai ei. Näin voidaan testata eli ”pollata” pääsyä mutexin suojaamalle kriittiselle alueelle ilman, että säie menee odotustilaan. (MSDN 2011i.)

Jos odotus päättyy aikakatkaisuun, WaitForSingleObject-funktio palauttaa virheen WAIT_TIMEOUT. Jos funktio palauttaa virheen WAIT_FAILED, GetLastError-funktiolla voidaan hakea tilanteeseen johtaneen virheen koodi. Jos objekti suljetaan kesken odotuksen, funktio palauttaa virheen WAIT_ABANDONED. Tämä on merkki siitä, että odotettava objekti on joutunut tilaan, jossa sitä ei voida enää käyttää. (MSDN 2011i.)

Kun WaitForSingleObject-funktio palauttaa arvon WAIT_OBJECT_0, se on saanut varattua mutex-objektin onnistuneesti. Tällöin mikään muu säie ei voi varata samaa objektia ennen kuin se on jälleen vapaa. (MSDN 2011i.)

```
1   DWORD WINAPI WaitForSingleObject (
2       __in HANDLE hHandle,
3       __in DWORD dwMilliseconds
4   );
```

KUVIO 9. WaitForSingleObject-funktion prototyyppi

Kun odotettavia objekteja on samanaikaisesti useita, odottamiseen voidaan käyttää WaitForMultipleObjects-funktiota, joka on esitelty kuviossa 10. Funktio toimii muuten samalla tavalla kuin WaitForSingleObject-funktio, mutta sille annetaan yksittäisen objektin sijasta objektilista. Lisäksi määritellään se, että odotetaanko

kaikkien objektien valmistumista kerralla, vai palataanko heti, kun ainakin yksi objekteista valmistuu. (MSDN 2011h.)

```
1   DWORD WINAPI WaitForMultipleObjects(
2       __in DWORD nCount,
3       __in const HANDLE *lpHandles,
4       __in BOOL bWaitAll,
5       __in DWORD dwMilliseconds
6   );
```

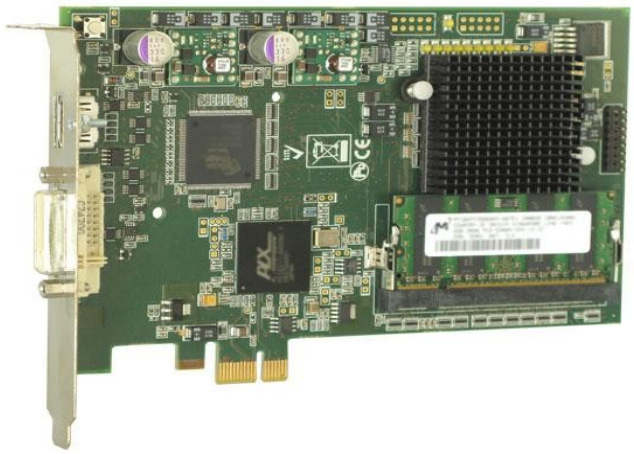
KUVIO 10. WaitForMultipleObject-funktion prototyyppi

6 UFG-JÄRJESTELMÄ

6.1 Tuotteet

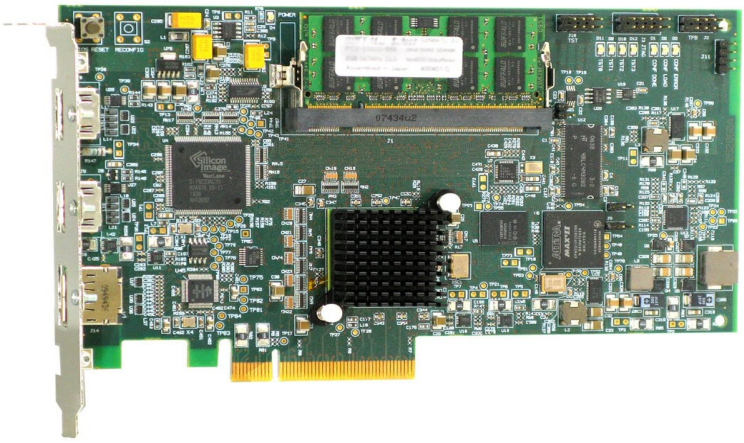
Kuvankaappauslaitteiden eli UFG-tuoteryhmän kehitys alkoi 2000-luvun alussa, jolloin Unigraf aloitti laivoissa käytettäväksi tarkoitetun mustan laatikon, Voyage Data Recorder (VDR), kehityksen. Ensimmäinen UFG-laite oli tarkoitettu analogisen tutkakuvan kaappaukseen. Sen laitteiston sekä ohjelmiston tuotekehityksen teki Unigrafin silloinen tytäryhtiö, unkarilainen Dexon Systems. Yhtiö on myöhemmin kehittänyt myös UFG-03- ja UFG-05-tuoteperheiden tuotteet.

Nykyisin UFG-tuoteryhmään kuuluu useita tuoteperheitä, joista UFG-03 ja sen seuraaja UFG-05 ovat jatkuvaan kuvankaappaukseen (streaming) soveltuvia tuotteita, ja UFG-04 on kehitetty analyysi- ja testaustyökaluksi. UFG-04-tuotteita on useita, osa yksikorttisia (kuvio 11) ja osa monikorttisia (kuvio 12). Kirjoittamishetkellä uusin, lokakuussa 2011 julkistettu ensimmäinen UFG-06 tuoteperheen tuote UFG-06 Nopea (kuvio 13), soveltuu sekä jatkuvaan kuvankaappaukseen että testaukseen, ja se on aiempia tuotteita nopeampi ja monipuolisempi.



KUVIO 11. Yksikorttinen UFG-04 HDMA -laite

KUVIO 12. Kaksikorttinen UFG-04 QLVDS -laite

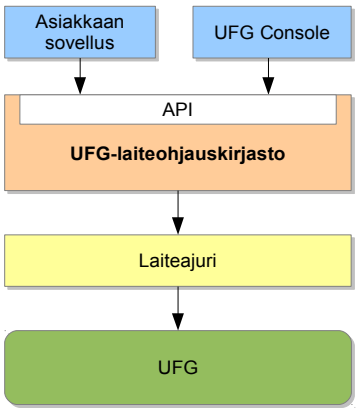


KUVIO 13. Uusi UFG-06 Nopea

UFG-04- ja UFG-06-laitteille on saatavilla sovelluskehityspaketti (SDK), joka tarjoaa käyttäjälle ohjelmointirajapinnan (API) räätälöityjen ohjaussovellusten toteuttamiseen. Unigrafin tavoitteena on tarjota monipuolinen ja joustava alusta testiautomaatio- ja tuotekehitystarkoituksia varten. Tämän vuoksi API:n on sovelluttava käytettäväksi yhtä hyvin skriptikielten (esim. Python, TCL) kuin ohjelmointikieltenkin (esim. C/C++, C#, Visual Basic, Delphi) kanssa. Sen on selviydyttävä mahdollisuuksien mukaan kaikista virhetilanteista häiritsemättä rajapintaa käyttävien ohjelmien toimintaa ja ennaltaehkäistävä ristiriitoja ja muita mahdollisia ongelmia.

6.2 Arkkitehtuuri

UFG-järjestelmä koostuu UFG-laitteesta, laiteajurista, ohjauskirjastosta ja sen sisältämästä ohjelmointirajapinnasta sekä sovelluksista, jotka käyttävät kyseistä rajapintaa. Sovellukset ovat joko Unigrafin omia tuotteita tai käyttäjän tekemiä räätälöityjä ohjelmia. Kuviosta 14 käy ilmi yhden Unigrafin ohjelmistotuotteen, UFG Consolen, suhde API:iin ja asiakkaan sovellukseen.



KUVIO 14. UFG-järjestelmän arkkitehtuuri

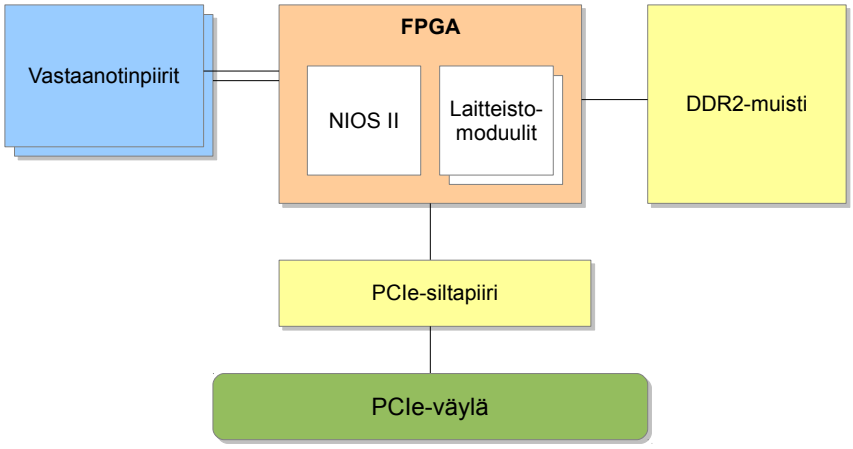
Opinnäytetyössä käsiteltävien tuoteperheiden laitteet tukevat yhteensä neljää eri videoliitäntästandardia. Nämä standardit ovat HDMI, DisplayPort, DVI ja LVDS. Taulukosta 1 voidaan nähdä eri laitteiden tukemat kunkin standardin mediatyypit.

TAULUKKO 1. UFG-laitteiden tukemat mediat

Tuote	Kortteja	Sisääntulot	Tuetut mediat
UFG-04 HDMA	1	1 x HDMI 1 x DVI	HDMI video HDMI audio DVI video
UFG-04 DP	1	1 x DisplayPort	DP video DP audio
UFG-04 LVDS	1	1- ja 2-kanavainen LVDS	LVDS video
UFG-04 QLVDS	2	1-, 2- ja 4-kanavainen LVDS	LVDS video
UFG-04 OLVDS	4	1-, 2-, 4- ja 8-kanavainen LVDS	LVDS video
UFG-06 Nopea	1	1 x DisplayPort 2 x HDMI	DP video DP audio HDMI video HDMI audio Audio return channel Ethernet over HDMI

6.3 Laitteisto

Fyysisesti UFG-laite on tietokoneen PCI Express -oheislaiteväylään liitettävä kortti, jonka pääkomponentteja ovat FPGA-piiri (Field Programmable Gate Array), DDR2-muisti, PCIe-siltapiiri (UFG-06:ssa integroitu FPGA-piiriin) ja yksi tai useampi videovastaanotinpiiri (kuvio 15). Laitteen kiinteänä osana on FGPA-piirin sisältämä Nios II -soft-prosessori, jossa on sovellusalustana µC/OS-II-reaaliaikakäyttöjärjestelmä. FPGA-piiri sisältää lisäksi laitteistomoduuleja, joilla analysoidaan ja muunnetaan kaapattavaa tietoa.



KUVIO 15. UFG-laitteen perusrakenne

Kuva- tai äänisignaali tulee sisään laitteeseen liittimeen kytketyn kaapelin kautta. Laitteessa signaali kulkee vastaanotinpiirille, joka muuttaa sen datavirraksi. Lisäksi vastaanotin kerää signaalista tila- ja mittaustietoja, joita se välittää eteenpäin pyydettäessä. FPGA-piirillä oleva tallennusjärjestelmä kerää datavirran puskurimuistiin, josta se siirretään DMA-tiedonsiirrolla (Direct Memory Access) tietokoneelle.

FPGA-piirillä olevia laitteistomoduuleja ohjataan lukemalla ja kirjoittamalla 32-bittisiä rekistereitä. Rekisterit näkyvät tietokoneohjelmalle muistikarttana (memory mapping), jossa jokaisella moduulilla on oma perusosoite (base address). Perusosoitteeseen lisätään rekisterin poikkeama (offset), jolloin saadaan rekisterin yksilöllinen osoite. Rekisterien kautta luetaan mm. kuvan ajoitukseen (timing) liit-

tyviä tietoja, CRC-tarkistussummia (Cyclic Redundancy Check) ja laitteen tilatietoja. Lisäksi niiden kautta kirjoitetaan asetuksia ja käytetään ohjaustoimintoja.

6.4 Ohjelmisto

UFG-järjestelmän ohjelmisto koostuu kolmesta eri tasosta, joista alimpana on laiteajuri (kuvio 14 sivulla 23). Ajuria käytetään avaamaan ja sulkemaan laite sekä alustamaan rekisterikartta ja DMA-tiedonsiirto. Laitteen kaikki muut toiminnot on toteutettu käyttämällä rekistereitä ja DMA-tiedonsiirtoa.

Laiteajuria käyttää laiteohjauskirjasto, joka sisältää ohjelmointirajapinnan. Rajapintaa käyttävät niin Unigrafin omat kuin asiakkaidenkin sovellukset. Kirjasto vastaa mm. laitteiden tunnistamisesta, resurssien ja tilatietojen ylläpidosta sekä ohjaustoimintojen välittämisestä käyttäjältä laitteelle.

7 LAITEOHJAUSKIRJASTON SUUNNITTELU JA TOTEUTUS

7.1 Työn lähtökohta ja vaatimusmäärittely

Ennen opinnäytetyön aloittamista UFG-04-tuoteperheen jokaiselle eri laitteelle oli olemassa oma ohjauskirjasto. Nämä kirjastot oli toteutettu C-kielellä. Niitä pystyi käyttämään lähes ainoastaan C- ja C++-kielisistä ohjelmista, koska funktioiden parametreina oli käytetty 8-bittisesti järjesteltyjä tietueita (byte-aligned structure). Kirjastojen rakenteesta johtuen niillä pystyi ohjaamaan useita laitteita yhtä aikaa, mutta yksittäisiä laitteita ei voinut käyttää säikeistetystä ohjelmasta. UFG-06-tuoteperheelle ei ollut olemassa ohjauskirjastoa lainkaan, koska ensimmäisen UFG-06-tuotteen kehittely aloitettiin vasta samoihin aikoihin opinnäytetyön kanssa.

Vanhojen kirjastojen ohjelmointirajapinnat poikkesivat toisistaan niin, että ne eivät olleet keskenään vaihdettavissa. Rajapintojen päälle oli rakennettu ”katto-DLL”, joka sisälsi varsinaisen asiakasohjelmointirajapinnan. Tähänkin ratkaisuun liittyi monia ongelmia. Pahin ongelmista oli se, että kyseistä rajapintaa ei kyetty säilyttämään yhteensopivana eri versioiden välillä. Tästä johtuen asiakkaat joutuivat päivittämään omia ohjelmiaan aina ottaessaan uutta versiota käyttöön.

Uudelle laiteohjauskirjastolle määritettyjä vaatimuksia olivat mahdollisuus ohjata useampaa kuin yhtä laitetta yhtä aikaa, mahdollisuus käyttää rajapintaa yhtäaikaisesti monesta ohjelmasta, mahdollisuus käyttää laitetta säikeistetysti, tuki kaikille UFG-04- ja UFG-06-tuoteperheiden laitteille sekä ohjelmointirajapinnan soveltuvuus erityyppisille ohjelmointikielille. Lisäksi oli toivottavaa, että rajapinta voitaisiin toteuttaa yhteen DLL-kirjastoon aikaisempien laitekohtaisten DLL-kirjastojen sijasta.

Kehitettävän laiteohjauskirjaston rakenteellisista vaatimuksista johtuen toteuttamistavaksi määriteltiin luokkapohjainen toteutus ja laiteläheisyyden vuoksi kieleksi valittiin C++. Alustaksi määritettiin Windows 2000, XP, Vista ja 7 sekä kolmen

jälkimmäisen 32- ja 64-bittiset versiot. Kehitystyökaluksi valittiin Embarcadero C++ Builder sen monipuolisen VCL-kirjaston vuoksi.

7.2 Käyttötapaukset

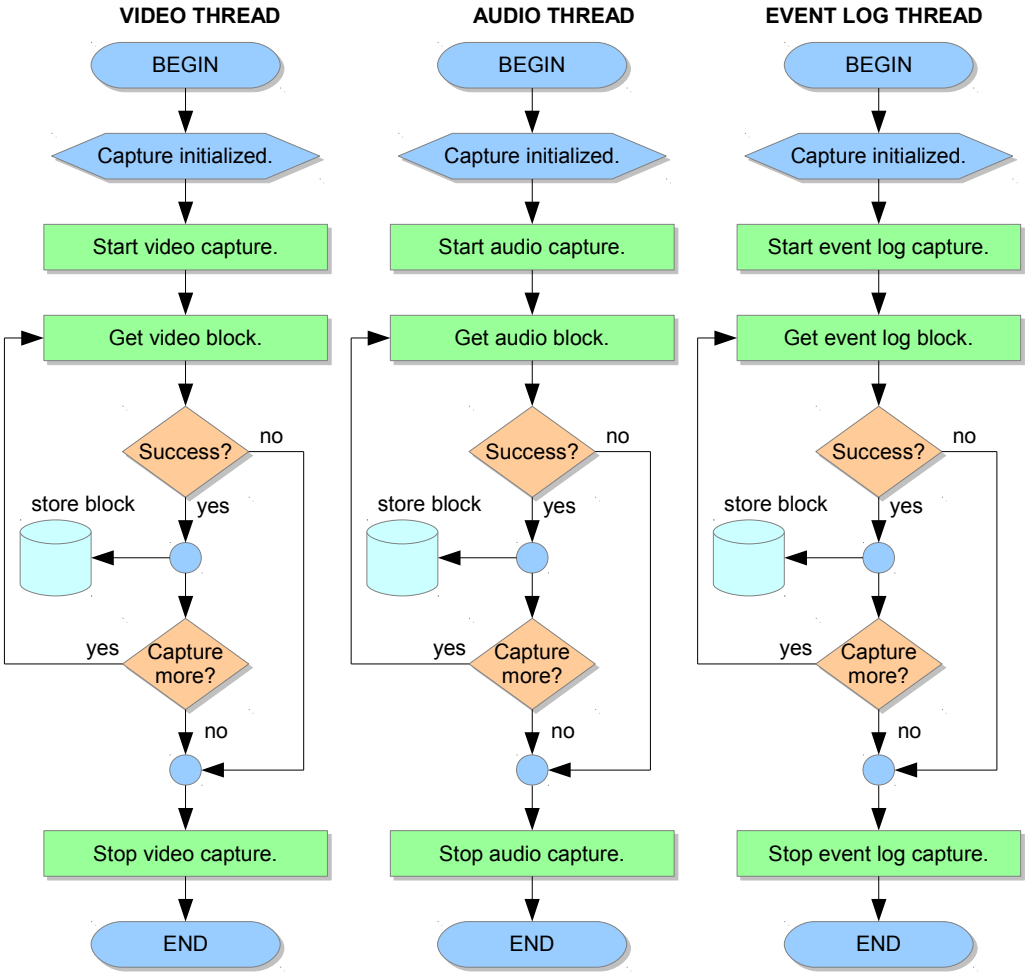
Asiakkaiden pääasiallisin tapa käyttää UFG-laitetta oli ollut yhdistetty kuvankaappaus ja signaalianalyysi. Jotkut asiakkaista olivat käyttäneet sitä pelkkään signaalin analysointiin, mutta näitä oli ollut suhteellisen vähän. Oli kuitenkin odotettavissa, että DisplayPort CTS tulisi yleistyessään lisäämään analysointi- ja mittauskäyttöä. Lisäksi oli odotettavissa, että mittaustietoja kerättäisiin lokiin monista lähteistä yhtä aikaa.

Edellä mainituista syistä johtuen järjestelmässä päätettiin ottaa huomioon se mahdollisuus, että kuvankaappauksen ollessa käynnissä yhdessä säikeessä toinen tai useampi säie keräisi samaan aikaan mittaustietoja. Lisäksi huomioitiin, että graafista käyttöliittymää käytettäessä käyttäjän tekemät muutokset tapahtuisivat millä hetkellä tahansa kuvankaappauksen ja tiedonkeruun aikana. Laite ei tee itse aloitteita ohjauskirjastoon päin, vaan tuottaa tietoa ainoastaan pyydettäessä, joten käyttötapaukset voitiin rajoittaa kirjaston ohjelmointirajapintaan.

Käyttötapauksia mietittäessä päädyttiin seuraaviin johtopäätöksiin ja linjauksiin:

- Kirjastoa voidaan käyttää monesta sovelluksesta yhtä aikaa.
- Laitetta voidaan käyttää vain yhdestä sovelluksesta kerrallaan. Laitteen tunnistetiedot ovat kuitenkin näkyvissä kaikille sovelluksille.
- Vaikka laitteessa olisi monta sisääntuloa, vain yhtä niistä voidaan käyttää kerrallaan. Tämä rajoitus johtuu UFG-laitteen sisäisistä rajoituksista.
- Yhdestä sisääntulosta voidaan kaapata videota, audiota ja/tai tapahtumalokia riippuen sisääntulon ominaisuuksista. Näistä ”lähteistä” voidaan kaapata dataa yhtäaikaisesti rinnakkaisissa säikeissä (kuvio 16).
- Kaappauksen aikana voidaan lukea ja kirjoittaa mitä tahansa mittaus- ja ohjausdataa. Tämä voi tapahtua periaatteessa rajoittamattomasta määrästä säikeitä.

Kuviossa 16 olevasta vuokaaviosta voidaan nähdä, että ulkoisesti video-, audio- ja tapahtumalokikaappauksella ei ole mitään tekemistä keskenään. Laitteessa on kuitenkin rajallinen määrä resursseja, joiden käyttöä joudutaan vuorottelemaan. Tämä vaikutti laiteohjauskirjaston toteutukseen siten, että vuorottelu jouduttiin toteuttamaan käyttämällä keskinäistä poissulkemista.



KUVIO 16. Rinnakkainen videon, audion ja tapahtumalokin kaappaus

7.3 Käsitteet

UFG-laitteesta puhuttaessa voidaan tarkoittaa joko yksikorttista laitetta (single card device) tai useamman kortin korttiryhmää (card array device). Nämä käsitteet on helppo sotkea keskenään, joten suunnitteluvaiheessa päätettiin määritellä käsitteet laitekokoonpano, kortti, fyysinen laite, sisääntulo ja lähde seuraavasti:

- Laite (device) tarkoittaa samaa kuin UFG-tuote. Laite voi koostua yhdestä tai useammasta kortista. Laitteesta käytetään myös nimitystä kokoonpano.
- Yhden kortin laitteita ovat UFG-04 LVDS, UFG-04 HDMI, UFG-04 DP ja UFG-06 Nopea.
- Usean kortin laitteita ovat UFG-04 QLVDS (kaksi UFG-04 LVDS -korttia) ja UFG-04 OLVDS (neljä UFG-04 LVDS -korttia).
- Kortti (card) on kirjaston välikerros, jonka kautta käytetään fyysistä laitetta. Yhden kortin kautta ohjataan yhtä fyysistä laitetta.
- Fyysinen laite (physical device) on tietokoneessa kiinni oleva kortti, joka on siis eri asia kuin kirjastossa käytetty *kortti*-käsite.
- Sisääntulo vastaa yhtä laitteessa olevaa vastaanotinpiiriä tai -piirisarjaa. Sisääntuloja voivat olla DP, HDMI, DVI ja eri LVDS-kombinaatiot.
- Lähde vastaa sisääntulosta kaapattavaa mediatyyppiä. Lähteitä voivat olla Video, Audio ja EventLog. Lähteitä tuetaan vaihtelevasti sisääntulon tyypistä riippuen.

7.4 Työn etenemisjärjestys

Laiteohjauskirjasto toteutettiin siten, että ensin laadittiin skenaariot käyttötapauksista ja sen perusteella suunniteltiin ohjelmointirajapinnan funktiot. Tämän jälkeen suunniteltiin kaikki rajapinnat, niiden väliset suhteet ja lukumäärät ja luotiin yleiskäyttöinen kehysrakenne eli framework. Kehysrakenne koostuu luokista, joista voidaan periyttää erikoistettuja luokkia kunkin laitteen tarpeisiin.

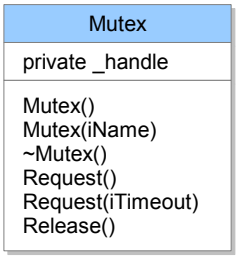
Erikoistuneita luokkia luotiin sitä mukaa kuin kehysrakenteen toteutus eteni. Samalla testattiin vaihe vaiheelta eri osien toimivuus. Yleiskäyttöisiä luokkia lisättiin kirjastoon sitä mukaa kuin tarvetta ilmeni. Kaikki osa-alueet syntyivät toistensa varassa ja joissakin jouduttiin tekemään muutoksia suunnitelmiin kesken työvaiheen.

Seuraavissa luvuissa esitellään ensin laiteohjauskirjaston sisäinen rakenne ja sitten ohjelmointirajapinta. Sisäisestä rakenteesta käydään ensin läpi yleiskäyttöiset luokat, sitten kehysrakenne yleisellä tasolla, minkä jälkeen siirrytään kehysrakenteen luokkakohtaiseen toteutukseen ja siitä periytettyihin erikoistettuihin luokkiin. Ohjelmointirajapinnan esittelyn yhteydessä käydään sekvenssikaavioiden tai sanallisen kuvauksen avulla läpi sitä, kuinka resurssien suojaus toteutettiin kirjastoon. Suojausta käsitellään osittain myös yleiskäyttöisten luokkien yhteydessä. Seuraavien lukujen järjestys ei ole kronologinen, vaan asiat esitellään mahdollisimman loogisessa ja toisiaan tukevassa järjestyksessä.

7.5 Kirjaston sisäinen rakenne

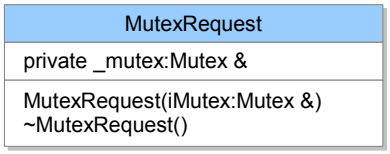
7.5.1 Mutex- ja MutexRequest-luokat

Resurssien suojausta varten laiteohjauskirjastoon toteutettiin Mutex-luokka (kuvio 17), joka piilottaa Windows-käyttöjärjestelmän mutex-toteutuksen ja helpottaa mutexin käyttöä C++-kielisessä ohjelmassa. Luokalle määriteltiin kaksi muodostinfunktiota (constructor), joista toinen luo nimeämättömän ja toinen nimetyn mutexin käyttöjärjestelmän CreateMutex-funktiolla. Lisäksi luokkaan toteutettiin Request-funktio mutexin varaamista ja Release-funktio vapauttamista varten. Request-funktio kutsuu käyttöjärjestelmän WaitForSingleObject-funkiota ja Release-funktio ReleaseMutex-funktiota. Tuhoajafunktio (destructor) kutsuu käyttöjärjestelmän CloseHandle-funktiota.



KUVIO 17. Mutex-luokka

Mutex-luokalle toteutettiin rinnakkaisluokka MutexRequest (kuvio 18), joka varaa mutexin muodostinfunktiossaan ja vapauttaa sen tuhoajafunktiossaan. Kuvion 19 esimerkkikoodista nähdään, kuinka mutexin varaus ja vapautus voidaan automatisoida luomalla MutexRequest-luokasta paikallinen objekti ja antamalla kyseinen mutex parametrina tämän muodostinfunktiolle.



KUVIO 18. MutexRequest-luokka

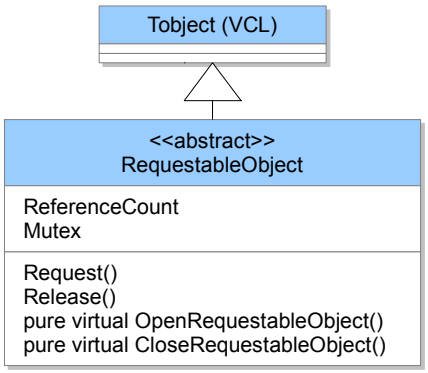
```
1   #include ”Class_Mutex.h”
2
3   class MyClass
4   {
5       public:
6           void MyProtectedFunction();
7
8       private:
9           Mutex _my_mutex;
10  };
11
12  #include ”Class_MutexRequest.h”
13
14  void MyClass::MyProtectedFunction()
15  {
17      // This requests the _my_mutex mutex.
19      MutexRequest m(_my_mutex);
20
22      // Do some critical section activity.
24      DoCriticalHwAccess();
25
27      // The _my_mutex mutex will be released automatically
28      // on exit within the release of the object m.
30  }
```

KUVIO 19. Esimerkki MutexRequest-luokan käytöstä

7.5.2 Viitelaskurijärjestelmä

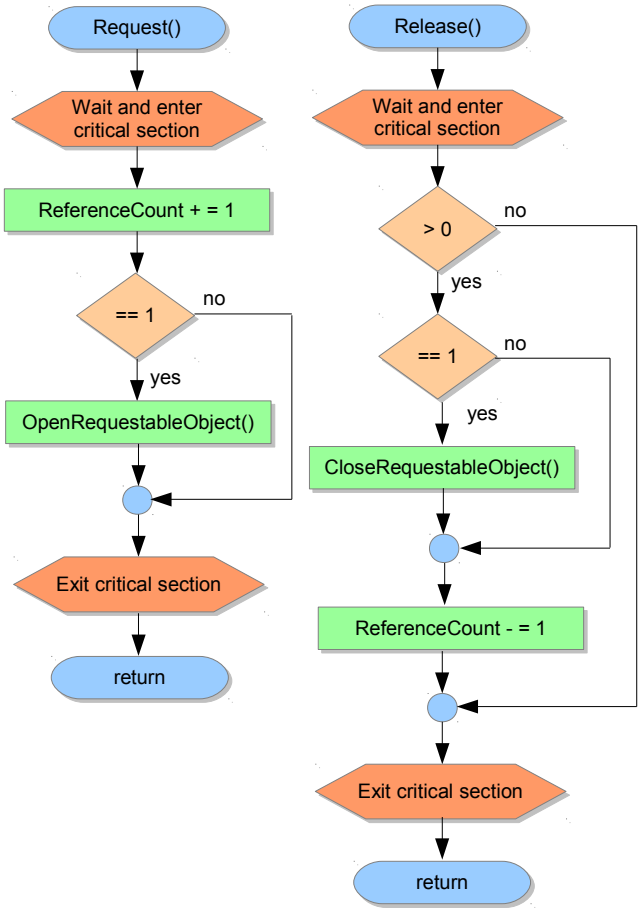
Laiteohjauskirjastossa tarvittiin erilaisia avattavia ja suljettavia objekteja, esimerkiksi laite, sisääntulo ja lähde. Näistä objekteista syntyi ketjuja, joissa useita johonkin tiettyyn objektiin liittyviä objekteja saattoi olla yhtä aikaa auki. Esimerkiksi yhdessä sisääntulossa saattoi olla monta lähdettä auki toisistaan riippumatta. Tästä johtuen avaamisen ja sulkemisen hallintaan tarvittiin viitelaskuri.

Viitelaskuria varten kirjastoon toteutettiin abstrakti luokka RequestableObject (kuvio 20), johon toteutettiin laskurin arvon sisältävä muuttuja ReferenceCount sekä funktiot Request ja Release. Funktioilla voidaan varata ja vapauttaa objekti, ja ne muuttavat laskurin arvoa vastaavasti. RequestableObject-luokasta periytettävien objektien avaamista ja sulkemista varten toteutettiin aidosti virtuaaliset funktiot OpenRequestableObject ja CloseRequestableObject, joita luokka kutsuu automaattisesti, jos viitelaskurin arvo nousee yli nollan tai laskee nollaan. Näin saatiin aikaan objektityyppi, jonka avulla voitiin avata ja sulkea laitteita ja niihin liittyviä muita ominaisuuksia hallitusti.



KUVIO 20. RequestableObject-luokka

Viitelaskuriluokan kutsumat virtuaalifunktiot suojattiin mutexilla kuvion 21 vuokaavioiden mukaisesti. Näin saatiin aikaan se, että kaikkien viitelaskuriluokasta periytettyjen objektien avaaminen ja sulkeminen ovat automaattisesti suojattuja. Samalla objekti sai käyttöönsä mutexin, jota se voi käyttää monien muidenkin toimintojensa suojaamiseen.



KUVIO 21. Request- ja release-operaatioiden vuokaaviot

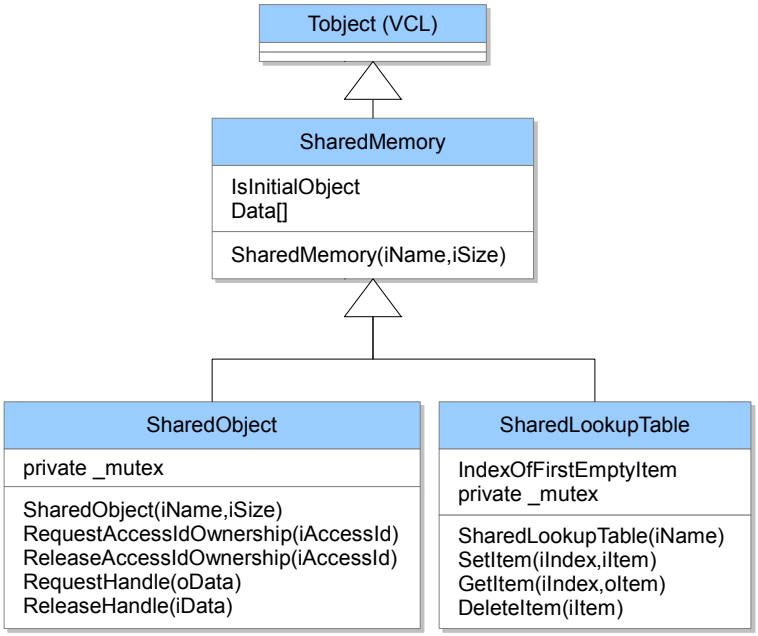
7.5.3 Tietojen jakaminen prosessien välillä

Yksi laiteohjauskirjaston vaatimuksista oli se, että kirjaston käytön on oltava mahdollista yhtä aikaa eri sovelluksissa. Tämän toteuttamista rajoitti se, että toinen laiteajureista esti yksittäisen laitteen avaamisen useasta sovelluksesta yhtä aikaa. Laitetta ei voitu avata tunnistusta varten, jos se oli jo auki jossain toisessa prosessissa. Tämä ongelma päätettiin ratkaista jakamalla laitetiedot eri prosessien välillä käyttämällä Windows-käyttöjärjestelmän jaettua muistia.

Muistin jakamista varten toteutettiin SharedMemory-luokka (kuvio 22), joka piilottaa Windows-käyttöjärjestelmän API-funktiot. Tästä luokasta periytettiin SharedObject-luokka, johon tehtiin tietojen jakamista varten tiedonvaihtoprotokolla funktioilla RequestHandle, ReleaseHandle, RequestAccessIdOwnership ja ReleaseAccessIdOwnership. Tavoitteena oli, että jokaisella laitteella olisi oma jaettu muistialue, johon laitteen tiedot tallennettaisiin, kun se avattaisiin ensimmäistä kertaa. Tämä mahdollistaisi myös laitetyyppikohtaiset erot jaettavissa tiedoissa.

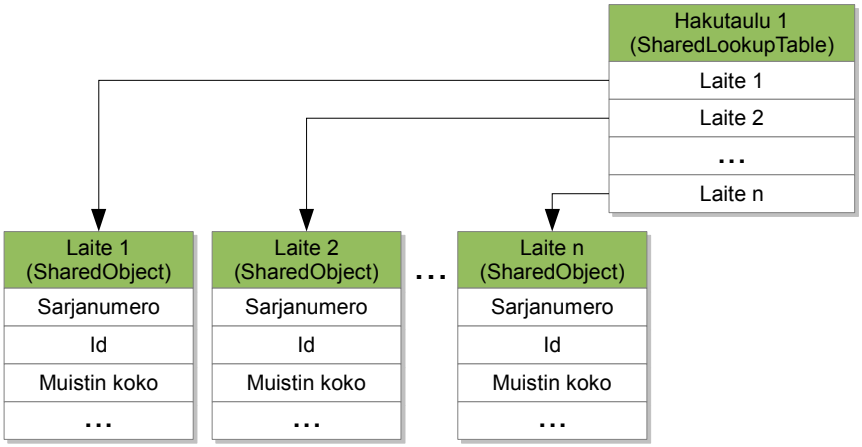
Kun jaettu objekti oli toteutettu, kohdattiin sen käyttöön liittyvä ongelma. Jotta jokaisen jaetun muistiobjektin nimi oli aina sama tietylle laitteelle, mutta uniikki eri laitteiden välillä, piti nimessä käyttää laitteen sarjanumeroa. Sarjanumero pystyttiin kuitenkin lukemaan vain laitteen ollessa auki. Ongelman ratkaisemiseksi luotiin hakutaulu SharedLookupTable, joka periytettiin SharedMemory-luokasta (kuvio 22).

Jotta tiedonvaihto SharedObject-objektien ja toisaalta SharedLookupTable-objektien välillä olisi prosessienvälisesti suojattua, molemmat objektityypit suojattiin käyttämällä nimettyä mutexia (kuvio 22). Nimetyn mutexin nimi muodostettiin objektista riippuen joko jaetun objektin tai jaetun hakutaulun nimen perusteella.



KUVIO 22. Jaettua muistia käyttävien luokkien luokkakaavio

Kun laite avataan ensimmäistä kertaa ja sille luodaan jaettu objekti, kyseisen objektin nimi tallennetaan ajurikohtaiseen hakutauluun (kuvio 23). Toinen kirjaston instanssi avaa hakutaulun ja muodostaa laitelistan sen avulla, jolloin sen ei tarvitse käyttää listan luomiseen laiteajuria tai avata laitteita.

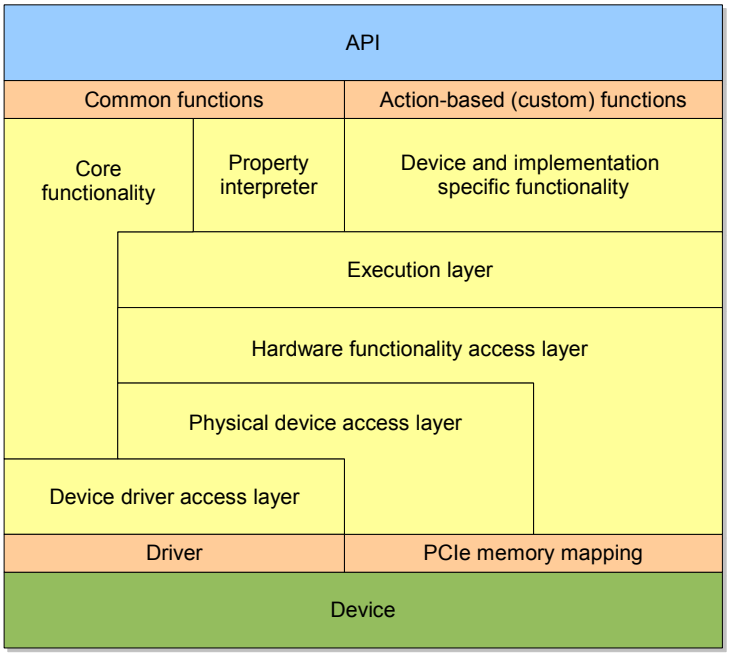


KUVIO 23. Jaetun muistin käyttö laitteiden tunnistetietojen jakamiseen

7.5.4 Kehysrakenne

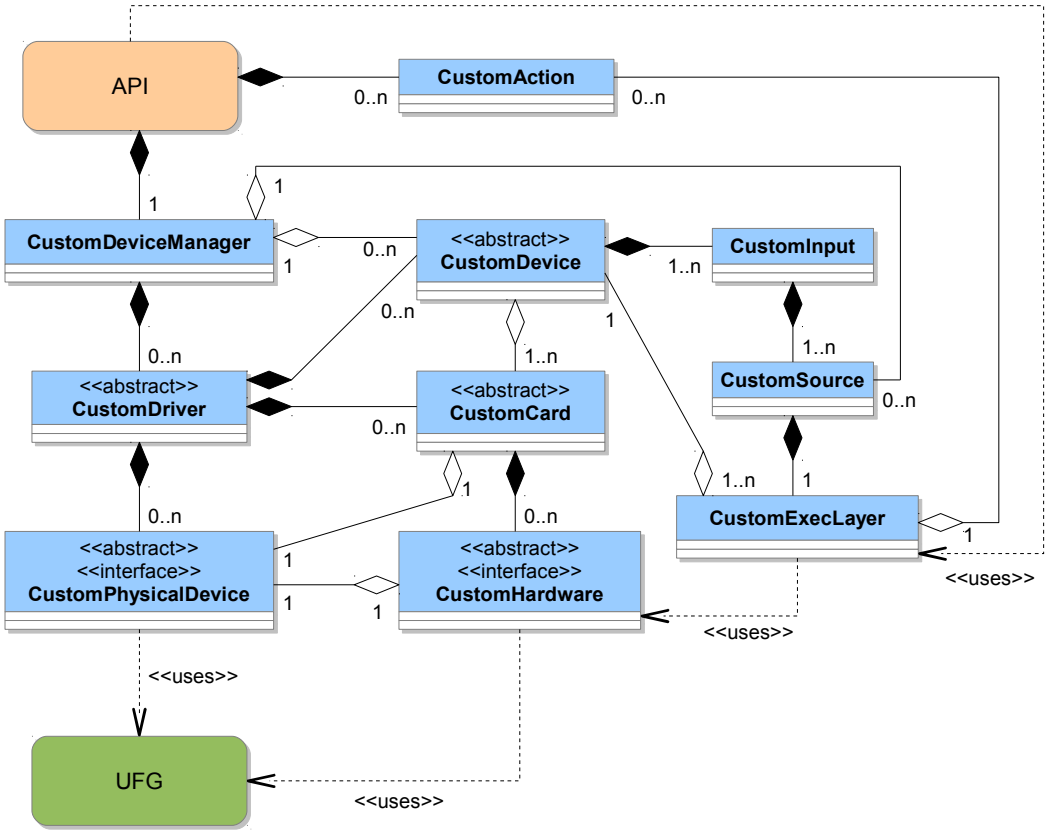
Kirjaston kehysrakenne syntyi osittain etukäteissuunnittelun, osittain toteutuksen aikana havaittujen tarpeiden pohjalta. Suunnitteluvaiheessa kehysrakenteen komponenttien nimeämiskäytännöstä päätettiin, että jokaisen kehysrakenteeseen kuuluvan kantaluokan nimen eteen tulee sana ”Custom” ja kantaluokan nimi alkaa heti tämän sanan jälkeen ilman alaviivaa. Kantaluokasta periytetyn luokan nimen etuliitteeksi tulee kantaluokan nimi ilman Custom-osuutta, ja luokan nimi erotetaan etuliitteestä alaviivalla.

Kirjaston toiminnallisesta rakenteesta piirrettiin kaavio, jossa toiminnot on kuvattu ylhäältä alaspäin etenevässä järjestyksessä (kuvio 24). Kaavio kertoo, miten API:ssa olevat funktiot jakaantuvat kahteen pääryhmään, ja millä tavalla nämä ryhmät liittyvät sisäisiin toimintoihin. Kaavion piirtäminen helpotti myöhemmin kehysrakenteen luokkien keskinäisten suhteiden määrittelyä, vaikka se ei ollutkaan täydellinen kuvaus kirjaston toiminnasta.



KUVIO 24. Laiteohjauskirjaston suunniteltu toiminnallinen rakenne

Kirjaston sisäisen rakenteen toteutuksesta piirrettiin luokkakaavio (kuvio 25), jota täydennettiin työn edetessä. Kaaviossa kaikki kehysrakenteen luokat yhdistettiin yhdeksi kokonaisuudeksi. Luokkien väliset muodostus-, koostumus- ja lukumäärä-suhteet pyrittiin mallintamaan mahdollisimman tarkasti, mutta joitakin toiminnan kuvauksen kannalta epäolennaisia yhteyksiä jätettiin pois kuvan selkeyttämiseksi. Kaaviosta jätettiin pois kaikki yleistykset, attribuutit ja operaatiot. Ne on kuvattu erikseen kunkin luokan kuvauksen yhteydessä.



KUVIO 25. Kehysrakenteen luokkakaavio

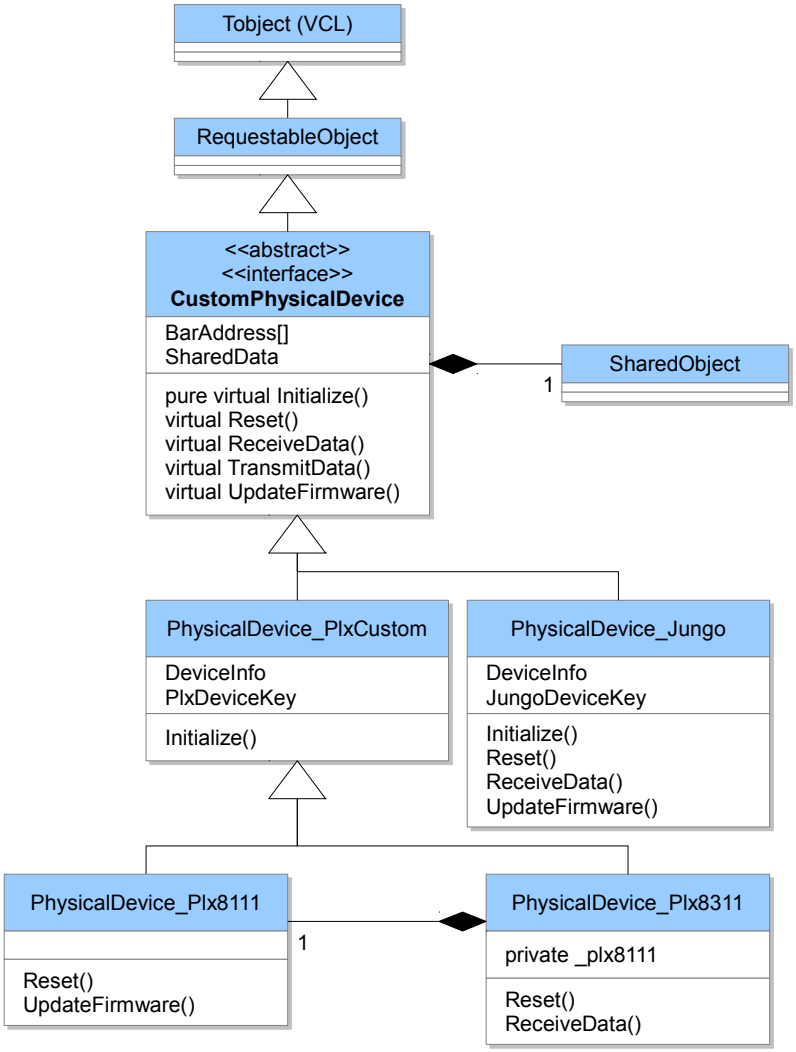
7.5.5 Ajurirajapinta

Laiteohjauskirjaston varsinaisen kehysrakenteen ja siitä erikoistettujen luokkien toteutus aloitettiin alimmalta tasolta, laiteajurin ohjelmointirajapinnasta. Kumpikin opinnäytetyössä käsiteltävä tuoteperhe käytti eri laiteajuria. Yhteistä näille ajureille oli se, että niiden rajapinta voitiin yleistää. Toisin sanoen kummassakin rajapinnassa oli toteutettu samat vähimmäisvaatimukset yhtenevän ohjauksen toteuttamiseksi:

- laitteiden listaaminen
- laitteen avaaminen ja sulkeminen
- laitteen tunnistustietojen haku
- PCIe-väylän BAR-osoitteiden kartoitus
- DMA-funktiot.

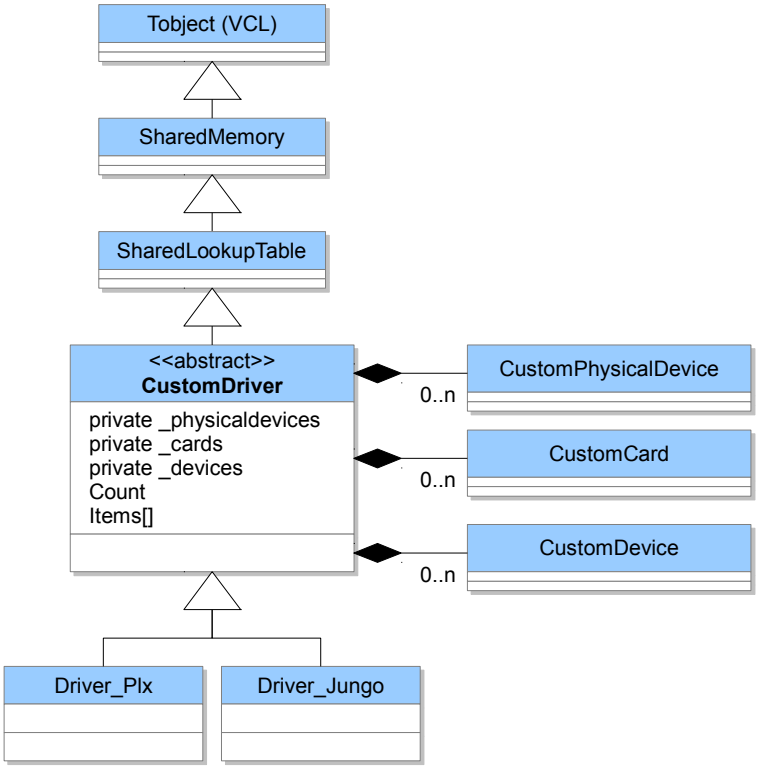
Ajurirajapintaa varten kehysrakenteeseen toteutettiin CustomPhysicalDevice-luokka, joka periytettiin RequestableObject-luokasta ja johon toteutettiin laitteen perustoiminnot tunnistusta, tiedonsiirtoa, BAR-osoitteiden hakua ja laitteistopäivitystä varten (kuvio 26). Luokka käyttää SharedObject-luokkaa laitteen tunnistustietojen jakamiseen prosessien välillä.

CustomPhysicalDevice-luokasta periytettiin luokat PhysicalDevice_PlxCustom ja PhysicalDevice_Jungo, joihin toteutettiin laitekohtaiset tunnistustietokentät ja funktiot. Koska UFG-04:n käyttämä PLX-siltapiiri koostuu todellisuudessa kahdesta erillisestä piiristä, PhysicalDevice_PlxCustom-luokasta periytettiin näitä varten luokat PhysicalDevice_Plx8111 ja PhysicalDevice_Plx8311. Näin molemmille piireille voitiin toteuttaa oma ohjausrajapinta. Yhdellä laitteella voi kuitenkin olla vain yksi rajapinta, joten PhysicalDevice_Plx8111-luokasta tehtiin osa PhysicalDevice_Plx8311-luokkaa.



KUVIO 26. Fyysisen laitteen luokkakaavio

Laitteiden listaamista varten kehysrakenteeseen toteutettiin abstrakti luokka CustomDriver (kuvio 27), josta periytettiin Driver_Plx-luokka UFG-04-tuoteperhettä ja Driver_Jungo-luokka UFG-06-tuoteperhettä varten. CustomDriver-luokan muodostinfunktio alustaa ajurin ja tekee listan tietokoneeseen kytketyistä UFG-laitteista. Se suorittaa fyysisten laitteiden etsimisen, tunnistamisen ja korttien luomisen sekä niiden yhdistelyn laitekokonaisuuksiksi. Objektit talletetaan niille varattuihin listoihin. Fyysisten laitteiden lista jaetaan muiden prosessien kanssa SharedLookupTable-isäntäluokan avulla.

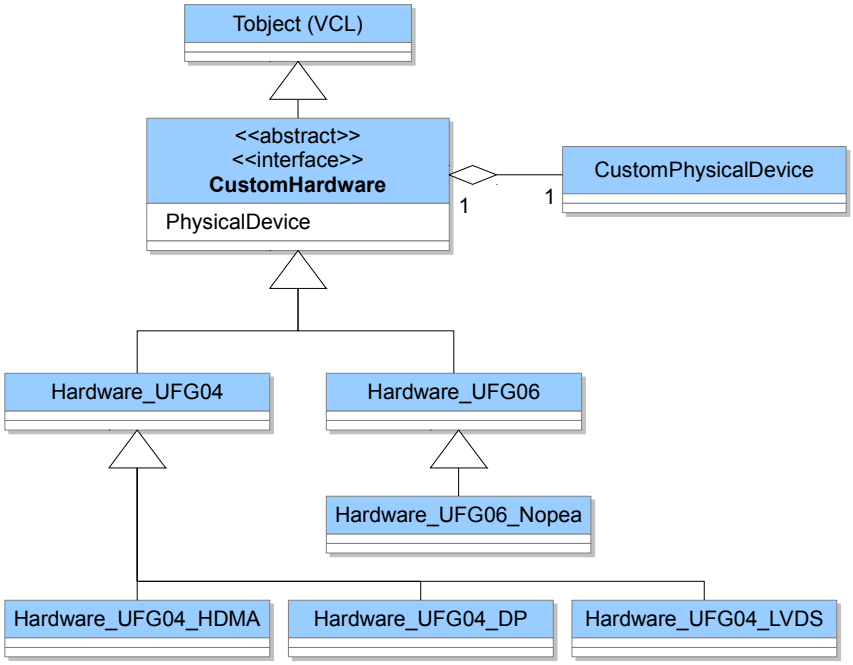


KUVIO 27. Laiteajuriluokkien luokkakaavio

7.5.6 Laitteistorajapinta

Fyysisen laitteen ohjausta varten toteutettiin abstrakti luokka CustomHardware (kuvio 28), joka sisältää osoittimen fyysisen laitteen objektiin. Luokan avulla päästään käsiksi fyysisen laitteen BAR-osoitekarttoihin ja funktioihin. CustomHardware-luokasta periytettiin jokaiselle tuoteperheelle oma luokka, joka toteuttaa kyseisen tuoteperheen laitteille yhteiset ominaisuudet. Näistä luokista periytettiin edelleen luokka jokaiselle eri laitetyypille näiden erityispiirteitä varten.

Jakamalla laitteistoluokat edellä mainitulla tavalla saavutettiin etua niin tyyppikohtaisten ominaisuuksien toteuttamisessa kuin myös suojauksessa. Laitteistokohtaisiin luokkiin oli helppo toteuttaa alimman tason suojaus sekä yksittäisten rekisterien tarkkuudella että toimintoryhmittäin. Yksittäisten rekisterien ja toimintojen suojaus ei kuitenkaan riittänyt, vaan korkeamman tason suojauksellekin todettiin olevan tarvetta.

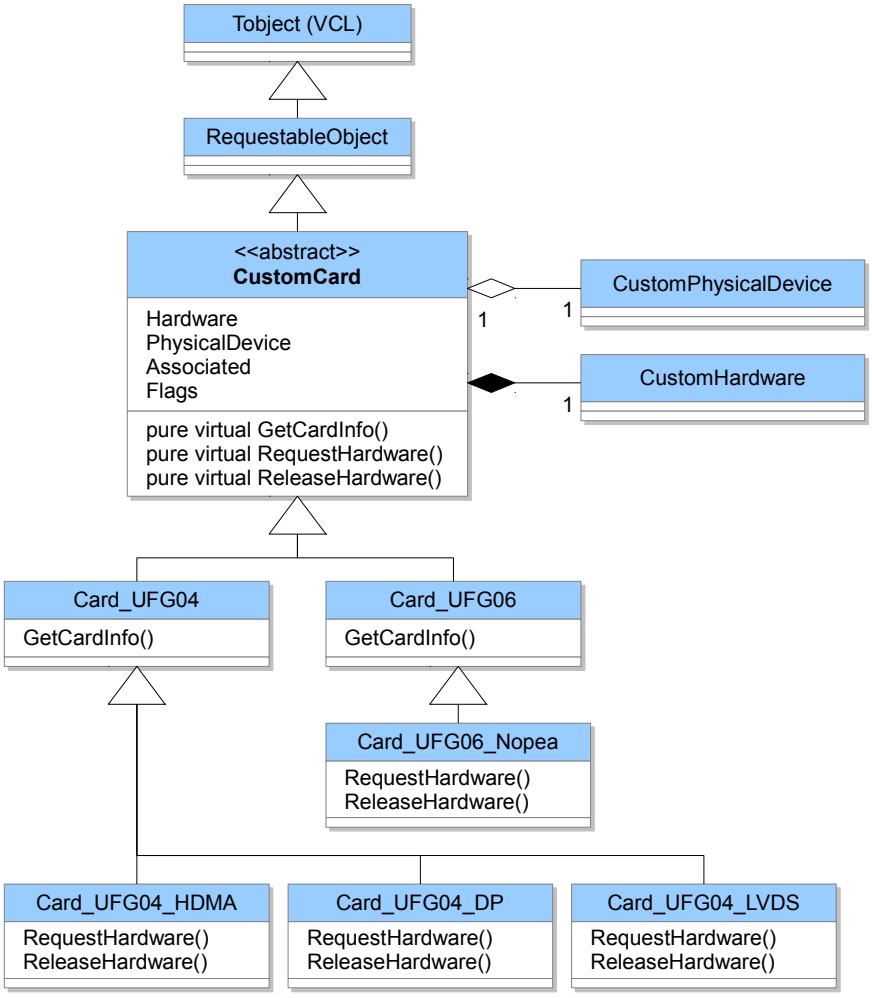


KUVIO 28. Laitteistoluokkien luokkakaavio

7.5.7 Korttivälikerros ja laite

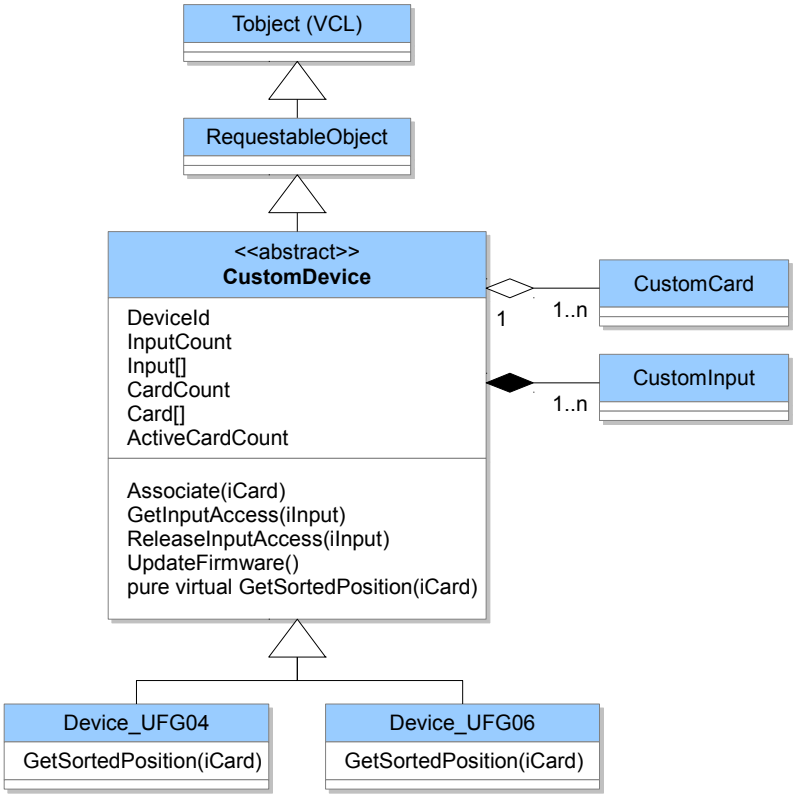
Korttivälikerroksen suunnittelussa todettiin, että kortin tulisi tarjota pääsy fyysiselle laitteelle sekä laitteistorajapinnan kautta että suoraan. Sen tulisi myös sisältää funktio, jolla voitaisiin lukea laitekohtaisia tunnistetietoja. Kirjastoon toteutettiin abstrakti luokka CustomCard, josta periytettiin laiteperhe- ja laitekohtaiset luokat (kuvio 29).

CustomCard-luokka oli yksi viitelaskuria hyödyntävistä luokista. Se omistaa CustomHardware-objektin, jonka se luo avaamisen (eli ensimmäisen varauspyynnön) yhteydessä ja tuhoaa suljettaessa (eli kun viimeinenkin varaus vapautetaan).



KUVIO 29. Korttivälikerroksen luokkakaavio

Laitekokoonpanolle luotiin luokka CustomDevice (kuvio 30), jossa toteutettiin yleiset laitetason toiminnot, esimerkiksi ohjelmistopäivitys. Luokkaan tehtiin lista laitteeseen assosioituja kortteja varten ja toteutettiin funktio, jolla kortteja voidaan assosioida laitteen kanssa. CustomDevice-luokasta periytettiin tuoteperhekohtaiset luokat, joissa toteutettiin tuoteperheiden erityispiirteitä, esimerkiksi menetelmä korttien järjestämiseksi sarjanumeron perusteella ja käyttölisenssien aktivointi. CustomDevice-luokka on myös yksi viitelaskuria käyttävistä luokista.



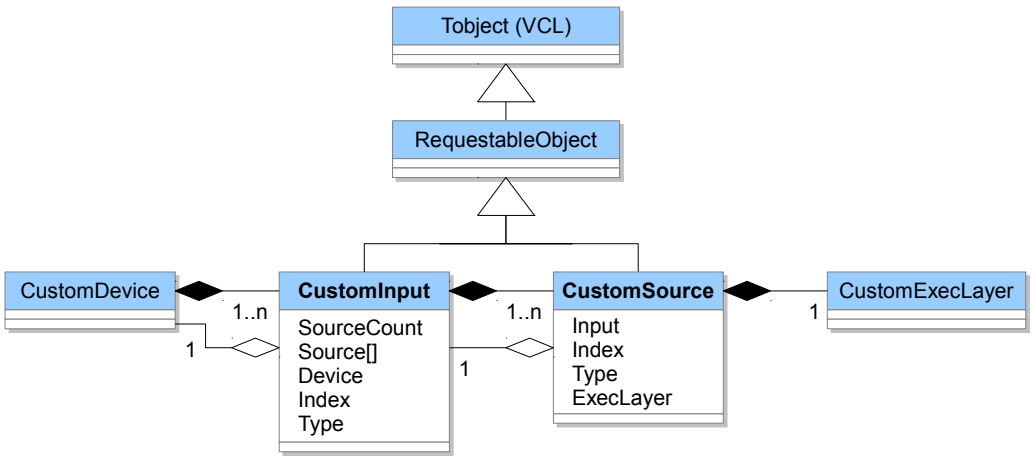
KUVIO 30. Laitekokoonpanon luokkakaavio

7.5.8 Sisääntulot ja lähteet

Laiteohjauskirjaston kokonaisrakennetta suunniteltaessa ajateltiin aluksi, että sisääntulolla on melko olematon rooli erillisenä objektina, ja tästä johtuen mietittiin mahdollisuuksia toteuttaa se lähteen yhteydessä tai jättää kokonaan toteuttamatta. Kun asiaa tarkemmin tarkasteltiin, kävi ilmi, että sisääntulo-objektilla on käsitteellisyydestään huolimatta huomattava merkitys lähdelistan, tyyppi-informaation ja aktiivisten korttien määrän säilönä. Lisäksi huomattiin, että erillisenä objektina se tarjoaa helpon tavan estää kahden sisääntulon yhtäaikainen käyttö ja siitä syntyvä laiteristiriita CustomDevice-luokan tasolla. Niinpä sisääntulolle luotiin oma luokka, CustomInput (kuvio 31). Luokka kelpasi käsitteellisen luonteensa vuoksi käytettäväksi sellaisenaan, eikä siitä tarvinnut periyttää muita luokkia.

Koska sisääntulo on laitteen ominaisuus, CustomInput määriteltiin CustomDevice-luokan jäseneksi. Yhdessä laitteessa voi olla monta sisääntuloa, joten CustomDevice-luokkaan toteutettiin lista niitä varten (kuvio 30 sivulla 44). Lisäksi todettiin, että laite voidaan avata, jos yksi sen sisääntuloista avataan. Niinpä sisääntulo määriteltiin avaamaan laite.

Sisääntulon avoimena olon todettiin olevan riippuvainen sen omistamien lähteiden aukiolojen määrästä, joten CustomInput-luokka periytettiin RequestableObject-luokasta viitelaskurin toteuttamiseksi. Lähde määriteltiin pyytämään avattaessa isäntäsisääntulonsa varaamista ja vapauttamaan varauksen suljettaessa. Yksi sisääntulo voi sisältää yhdestä kolmeen lähdettä. Koska lähde on se, josta dataa kaapataan, sille luotiin oma luokka, CustomSource (kuvio 31). CustomInput-luokkaan toteutettiin lista lähteitä varten.



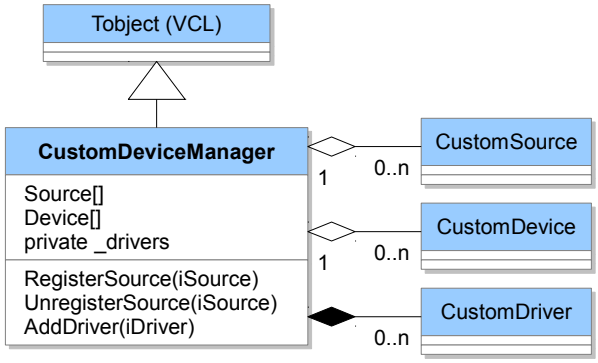
KUVIO 31. Sisääntulon ja lähteen luokkakaavio

7.5.9 Järjestelmänhallinta

Laitelistaa, avoimien lähteiden listaa, lähteiden hallintafunktioita ja ajurilistaa varten luotiin järjestelmähallintaluokka CustomDeviceManager (kuvio 32). Luokan muodostinfunktioon toteutettiin edellä mainittujen listojen luominen ja alustus ja tuhoamisfunktioon listojen tyhjennys ja vapautus. Luokan rajapintaan toteutettiin

AddDriver-funktio, jolla luokan objektille voidaan lisätä CustomDriver-tyyppisiä laiteajuriobjekteja. Funktio käy hakemassa laiteajuriobjektilta kaikki ajurin listaamat laitteet ja tallettaa laiteobjektit järjestelmähallintaobjektin sisäiseen listaan. Näin saatiin aikaan yhdessä paikassa oleva laitelista, joka sisältää kaikkien laiteajuriobjektien löytämät laitteet.

Järjestelmähallintaluokkaan tehtiin metodi, jolla voidaan rekisteröidä lähde avoimeksi tai poistaa suljettavan lähteen rekisteröinti. Tällä voitiin toteuttaa yksinkertaisesti ja keskitetysti lähteiden avoimena olon ja lähdeosoittimien oikeellisuuden tarkistus.



KUVIO 32. Järjestelmänhallintaluokan luokkakaavio

7.5.10 Suorituskerros

Koko kehysrakenteen työläin ja vaativin osuus oli suorituskerros, joka on komentotulkki ja tietosäilö lähdekohtaisille ominaisuuksille. Suorituskerros kokoaa kaikki edellä käsitellyt kehysrakenteen osa-alueet kokonaisuudeksi ja integroi ne ohjelmointirajapinnan kanssa.

Suorituskerroksen ydinkomponentiksi luotiin CustomExecLayer-kantaluokka, josta periytettiin kaikki laite-, input- ja lähdekohtaiset ExecLayer-luokat. Taulukossa 2 esitetään, mihin laitteeseen, sisääntuloon ja lähteeseen mikäkin luokka liittyy. Taulukossa esiintyviä luokkia ei esitetä opinnäytetyössä luokkakaaviona niiden lu-

kumäärän vuoksi. Luokkien periytymissuhteet käyvät ilmi taulukon Kantaluokka-sarakkeesta.

TAULUKKO 2. Suorituskerroksen luokat

Luokka	Kantaluokka	Laite	Sisääntulo	Lähde		
				Video	Audio	EventLog
ExecLayer_UFG04	CustomExecLayer	Kaikki	Kaikki	X	X	X
ExecLayer_UFG04_Video	ExecLayer_UFG04	Kaikki	Kaikki	X		
ExecLayer_UFG04_DP_Video	ExecLayer_UFG04_Video	UFG-04 DP	DP	X		
ExecLayer_UFG04_DVI_Video	ExecLayer_UFG04_Video	UFG-04 HDMA	DVI	X		
ExecLayer_UFG04_HDMI_Video	ExecLayer_UFG04_Video	UFG-04 HDMA	HDMI	X		
ExecLayer_UFG04_SLVDS_Video	ExecLayer_UFG04_Video	UFG-04 LVDS	SLVDS	X		
ExecLayer_UFG04_DLVDS_Video	ExecLayer_UFG04_SLVDS_Video	UFG-04 LVDS	DLVDS	X		
ExecLayer_UFG04_QLVDS_Video	ExecLayer_UFG04_SLVDS_Video	UFG-04 QLVDS	QLVDS	X		
ExecLayer_UFG04_OLVDS_Video	ExecLayer_UFG04_QLVDS_Video	UFG-04 OLVDS	OLVDS	X		
ExecLayer_UFG04_D2LVDS_Video	ExecLayer_UFG04_SLVDS_Video	UFG-04 QLVDS	D2LVDS	X		
ExecLayer_UFG04_Q2LVDS_Video	ExecLayer_UFG04_D2LVDS_Video	UFG-04 OLVDS	Q2LVDS	X		
ExecLayer_UFG04_Audio	ExecLayer_UFG04	UFG-04 DP, UFG-04 HDMA	DP, HDMI		X	
ExecLayer_UFG04_EventLog	ExecLayer_UFG04	Kaikki	Kaikki			X
ExecLayer_UFG06	CustomExecLayer	Kaikki	Kaikki	X	X	X
ExecLayer_UFG06_Video	ExecLayer_UFG06	Kaikki	Kaikki	X		
ExecLayer_UFG06_DP_Video	ExecLayer_UFG06_Video	UFG-06 Nopea	DP	X		
ExecLayer_UFG06_HDMI_Video	ExecLayer_UFG06_Video	UFG-06 Nopea	HDMI 1, HDMI 2	X		
ExecLayer_UFG06_Audio	ExecLayer_UFG06	Kaikki	Kaikki		X	
ExecLayer_UFG06_EventLog	ExecLayer_UFG06	Kaikki	Kaikki			X

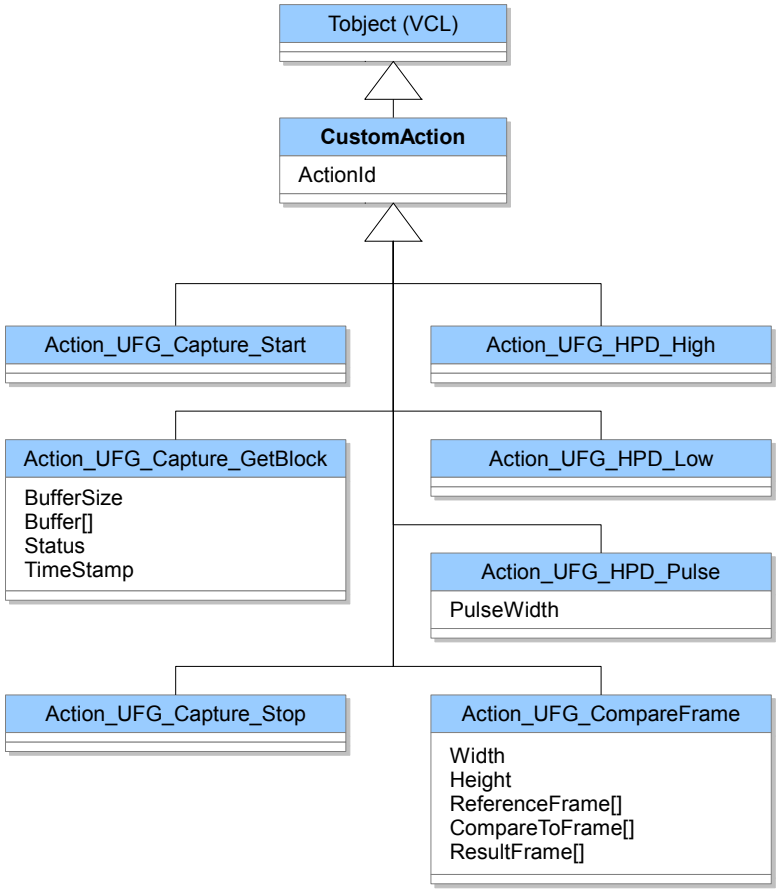
Suorituskerroksen luokissa toteutettiin kaikki erot, joita eri laitteiden, sisääntulojen ja lähteiden välillä voi olla. Monikerroksisen toteutuksen ansiosta yleisemmät toiminnot voitiin toteuttaa ylemmissä luokissa ja yksityiskohtaisemmat erot alemmissa luokissa. Eri tuoteperheiden välillä olevia yhtäläisyyksiä ei kuitenkaan voitu tällä menetelmällä yleistää, vaan osa toiminnoista jouduttiin toteuttamaan kahteen eri paikkaan täsmälleen samalla tavalla. Tämä ongelma päätettiin kuitenkin rajata opinnäytetyön ulkopuolelle ja jättää jatkokehittelyn aiheeksi.

Suorituskerroksen luokkiin toteutettiin kaikki ne laajemmat toimintokohtaiset suojausominaisuudet, joiden tarve havaittiin laitteistokerroksen kehittämisen yhteydessä. Koska suorituskerroksen toteutus on suurimmaksi osaksi lähdekohtainen, tällä saavutettiin se etu, että yksittäisen lähteen suojaus ei häiritse toisten lähteiden samanaikaista käyttöä, ellei lähteillä ole yhteisiä alemman tason resursseja.

7.5.11 Erikoistuneet funktiot

Erikoistuneilla funktioilla tarkoitetaan tässä yhteydessä niitä funktioita, jotka eivät ole ohjelmointirajapinnan kannalta yleiskäyttöisiä, vaan liittyvät enemmän tai vähemmän ohjattavan laitteen ominaisuuksiin tai luonteeseen. Näitä funktioita varten tarvittiin suorituskerroksen ja ohjelmointirajapinnan välille mekanismi, jolla vältetään erikoistoimintojen lisäämisen tai poistamisen vaikutus suorituskerroksen kantaluokkaan.

Tarkoitusta varten luotiin menetelmä, jossa funktiokutsusta muodostetaan objekti. Tämä objekti välitetään rajapintojen läpi suorittavalle tasolle asti, jolloin vain kutsujan (ohjelmointirajapinta) ja suorittajan (funktion toteuttava luokka suorituskerroksessa) tarvitsee tuntea funktion toteutus, ja muut rajapinnat ja välikerrokset välittävät objektin eteenpäin tietämättä, mitä se sisältää. Näin ollen myös funktion suojaus voidaan rajoittaa sen toteutukseen ja välikerrosten ei tarvitse ottaa siihen kantaa. Funktio-objektia varten luotiin luokka CustomAction, josta periytettiin luokat jokaiselle ohjelmointirajapinnan erikoistuneelle funktiolle (kuvio 33).



KUVIO 33. Erikoistuneiden funktio-objektien luokkakaavio

7.6 Ohjelmointirajapinnan funktiot ja suojaus

7.6.1 Suunnitteluperiaatteet

Ohjelmointirajapinta pyrittiin suunnittelemaan siten, että se olisi mahdollisimman yhtenäinen, suppea ja helppo omaksua. Lisäksi sen suunnittelussa pyrittiin ottamaan huomioon se, että rajapintaa tultaisiin käyttämään useilla erilaisilla ohjelmointikielillä. Nämä seikat huomioon ottaen rajapinnan funktioille laadittiin seuraavat säännöt:

- Paluuarvon tyyppi on joko 32-bittinen etumerkillinen kokonaisluku tai ei mitään (void).
- Negatiivinen paluuarvo on aina virhekoodi.
- Nolla tai positiivinen paluuarvo on funktion onnistunut tulos.
- Jos funktio palauttaa muita arvoja kuin yhden 32-bittisen positiivisen kokonaisluvun, se kirjoittaa arvot osoitinparametreina annettuihin muuttujiin.
- Funktion parametri ei voi olla tietue eikä tietueen osoite.
- Funktion nimi alkaa aina UFGAPI_-etuliitteellä. Ainoana poikkeuksena sallitaan GetDLLVersion, joka on samanniminen lähes kaikissa Unigrafin DLL-tiedostoissa.
- Funktion nimi sisältää ryhmän, johon funktio kuuluu. Esimerkiksi UFGAPI_Device_-alkuinen funktio kuuluu laitehallintafunktioiden ryhmään. Ryhmän nimi tulee aina heti UFGAPI_-etuliitteen jälkeen. Kaikilla funktioilla ei ole ryhmää.

Yllä olevista säännöistä eniten ohjelmointirajapinnan ja kirjaston suunnitteluun vaikutti tietueiden ja tietueen osoitteiden käyttökielto funktioiden parametreina. Tämä sääntö määriteltiin, koska tietueiden käsittely eroaa eri ohjelmointikielten välillä niin paljon, että jokin toteutustapa olisi aina sulkenut pois jonkin ohjelmointikielistä. Sääntö johti siihen, että rajapintaan suunniteltiin erikseen funktiot

tietueiden käsittelyä varten. Nämä funktiot muodostavat yhdessä rajapinnan nimeltä Property Interface.

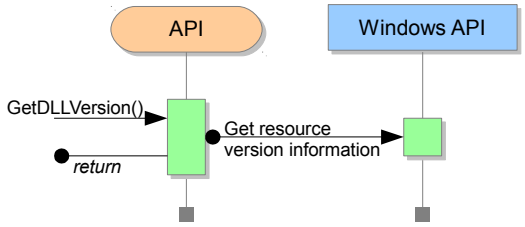
7.6.2 Kirjaston käyttöönotto ja alustus

Kun kirjasto on ladattu muistiin, käyttöönoton ensimmäinen vaihe on versionumeron haku. Versionumero on kolmiosainen lukusarja, joka koostuu pääversiosta (major version number), alaversiosta (minor version number) ja uudistusnumerosta (revision number). Versionumerolla on merkittävä rooli taaksepäin yhteensopivuuden kannalta. UFG-järjestelmässä pääversion muuttuminen tarkoittaa yhteensopivuuden katkeamista, ja sitä pyritään välttämään kaikin keinoin. Alaversion muuttuminen tarkoittaa, että ohjelmointirajapintaa on laajennettu uusilla ominaisuuksilla, mutta yhteensopivuus taaksepäin säilyy. Uudistusnumeroa muutetaan, kun ohjelmointirajapinta ja ulospäin näkyvä käyttäytyminen pysyvät muuttumattomina, eli ainoastaan kirjaston sisäisiä ominaisuuksia on korjattu tai muutettu.

Versionumeron hakemiseksi toteutettiin funktio GetDLLVersion, jonka prototyyppi on esitelty kuviossa 34. Tätä funktiota voidaan käyttää, vaikka kirjastoa ei olisi alustettu. Näin varmistettiin, että ainakin yksi funktio on kaikissa versioissa sama, ja kirjastoa käyttävät ohjelmat voivat aina lukea vähintään versionumeron, mikäli DLL-kirjaston tiedosto ei ole vaurioitunut. GetDLLVersion-funktio käyttää eräitä käyttöjärjestelmän ominaisuuksia versionumeron hakuun, mutta se todettiin vapaakäyntiseksi, joten kriittisen alueen suojausta ei tarvittu (kuvio 35).

```
1   INT32 WINAPI GetDLLVersion(
2       PINT32 oMajor,
3       PINT32 oMinor,
4       PINT32 oRevision
5   );
```

KUVIO 34. Versionhakufunktion prototyyppi



KUVIO 35. Versionhakufunktion sekvenssikaavio

Versionumeron hakemisen ja yhteensopivuuden varmistamisen jälkeen kirjasto voidaan alustaa. Windows-käyttöjärjestelmä kutsuu DLL-kirjaston latauksen ja vapautuksen yhteydessä kirjaston pääfunktiota (DllMain), jossa voidaan tehdä alustuksia. Laiteohjauskirjaston järjestelmähallinnan varaus ja vapautus olisi voitu periaatteessa tehdä pääfunktiossa, mutta käyttöjärjestelmä asettaa sen käytölle rajoituksia. Yksi rajoituksista on, että pääfunktion suorituksen tulisi mahdollisimman lyhyt, jotta se ei häiritse käyttöjärjestelmän muita toimintoja.

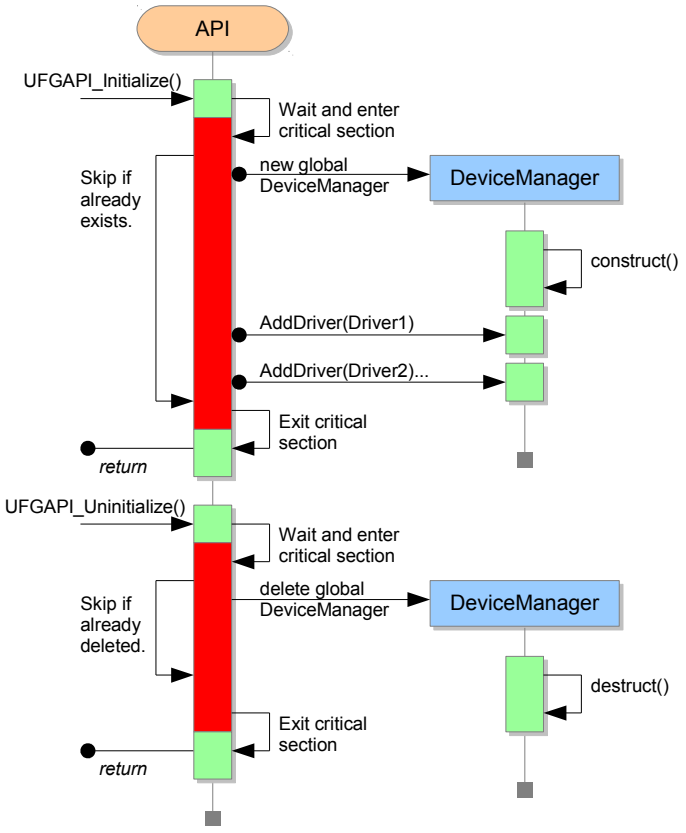
Käyttöjärjestelmän lisäksi ohjelmointi- ja skriptikielet asettavat kirjaston alustukselle omia vaatimuksia. Monet korkean tason ohjelmointi- ja skriptikielet (esim. C#, VB.NET, Python) käyttävät ajonaikaisia tulkkausjärjestelmiä, jolloin kirjaston lataamis- ja vapauttamishetki eivät ole käyttäjän määriteltävissä samalla tavalla kuin C- ja C++-kielissä.

Edellä mainituista syistä alustusta ei voitu toteuttaa pääfunktiossa, vaan ohjelmointirajapintaan toteutettiin sitä varten erilliset funktiot. Nämä funktiot ovat UFGAPI_Initialize ja UFGAPI_Uninitialize, joiden prototyypit on esitelty kuviossa 36. UFGAPI_Initialize-funktio luo objektit kirjaston järjestelmähallintaluokasta ja laiteajuriluokista, jotka puolestaan tekevät listan UFG-laitteista ja alustavat ne käyttöä varten. UFGAPI_Uninitialize-funktio tuhoaa järjetelmähallintaobjektin, jolloin sen tuhoajafunktio vapauttaa kaikki varatut resurssit.

```
1   INT32 WINAPI UFGAPI_Initialize();
2
3   VOID WINAPI UFGAPI_Uninitialize();
```

KUVIO 36. Kirjaston alustusfunktioiden prototyypit

Järjestelmähallintaobjekteja voi olla vain yksi yhtä kirjaston instanssia kohden. Niinpä UFGAPI_Initialize-funktio toteutettiin siten, että sitä voidaan kutsua useita kertoja ilman järjestelmähallintaobjektin varaamista uudelleen. Koska järjestelmähallintaobjekti on kirjaston sisällä globaali, funktion toiminta suojattiin mutexilla, etteivät samanaikaiset funktiokutsut johtaisi objektin monistumiseen (kuvio 37). UFGAPI_Uninitialize-funktion kanssa meneteltiin vastaavalla tavalla päällekkäisten muistinvapautusten estämiseksi. Valittu toteutustapa noudattaa singleton-suunnittelumallin periaatetta.



KUVIO 37. Alustusfunktioiden sekvenssikaavio

7.6.3 Laitehallinta

Laiteohjauskirjaston alustuksen yhteydessä luodaan lista tietokoneeseen kytketyistä UFG-laitteista. Samalla jokaiselle laitteelle haetaan tunnistetietoja, esimerkiksi sarjanumero, laitenimi ja versionumerot. Lisäksi luodaan listat laitteen tukemista sisääntuloista ja niiden tukemista lähteistä.

Ohjelmointirajapintaan toteutettiin funktio jokaisen edellä mainitun listan läpikäymiseen. Laitelista voidaan hakea UFGAPI_Device_Enum-funktiolla (kuvio 38, rivit 1-4), laitteen sisääntulojen lista UFGAPI_Device_EnumInputs-funktiolla (kuvio 38, rivit 6-10) ja sisääntulon lähdelista UFGAPI_Device_EnumSources-funktiolla (kuvio 38, rivit 12-17). Listat toteutettiin alun perin vapaakäyntisiksi, joten niitä ei tarvinnut suojata. Toimintaperiaate on yksinkertainen: funktiot palauttavat kunkin objektin tyyppikoodin indeksin perusteella, mikäli indeksi osoittaa olemassa olevaan objektiin.

```
1   INT32 WINAPI UFGAPI_Device_Enum(
2       INT32 iDeviceIndex,
3       PINT32 oDeviceId
4   );
5
6   INT32 WINAPI UFGAPI_Device_EnumInputs(
7       INT32 iDeviceIndex,
8       INT32 iInputIndex,
9       PINT32 oInputType
10  );
11
12  INT32 WINAPI UFGAPI_Device_EnumSources(
13      INT32 iDeviceIndex,
14      INT32 iInputIndex,
15      INT32 iSourceIndex,
16      PINT32 oSourceType
17  );
```

KUVIO 38. Listojen läpikäymisfunktioiden prototyypit

UFG-laite voi koostua yhdestä tai useammasta kortista, joista jokaiselle on olemassa erikseen tunnistetiedot, kuten sarjanumero, versionumero ja puskurimuistin koko. Jotta tunnistetiedot voitaisiin hakea jokaiselta kortilta erikseen, ohjelmointirajapintaan toteutettiin funktiot UFGAPI_Device_GetCardCount (kuvio 39, rivit 1-4) ja UFGAPI_Device_GetCardInfo (kuvio 39, rivit 6-12). Tunnistetietojen ha-

kufunktiot ovat listojen läpikäymisfunktioiden tapaan vapaakäyntisiä, joten ne eivät tarvitse suojausta.

```
1   INT32 WINAPI UFGAPI_Device_GetCardCount(
2       INT32 iDeviceIndex,
3       PINT32 oCardCount
4   );
5
6   INT32 WINAPI UFGAPI_Device_GetCardInfo(
7       INT32 iDeviceIndex,
8       INT32 iCardIndex,
9       INT32 iInfo,
10      INT32 iSize,
11      PVOID oData
12  );
```

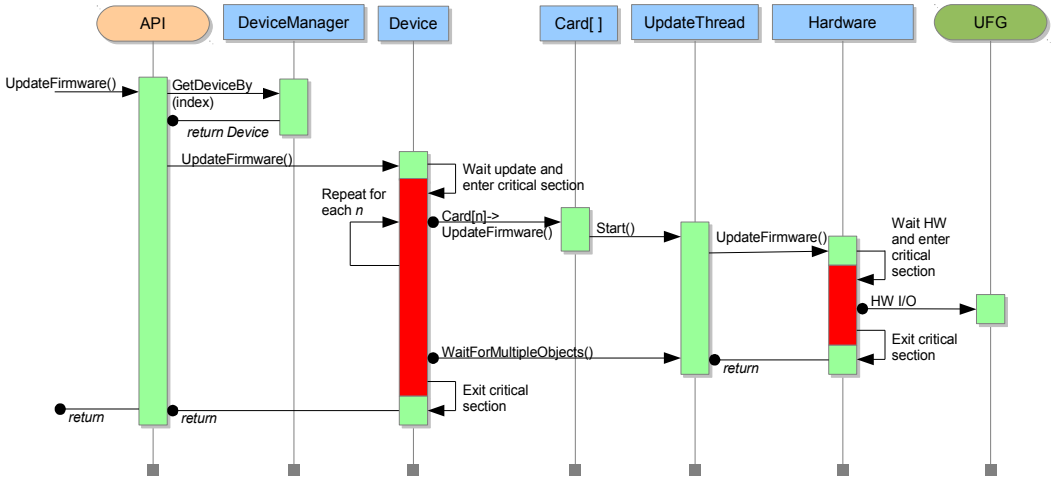
KUVIO 39. Tunnistetietojen hakufunktioiden prototyypit

UFG-laitteen ohjelmistopäivityksiä varten rajapintaan lisättiin funktio UFGAPI_Device_UpdateFirmware (kuvio 40). Funktiolle annetaan parametrilla iFileName (kuvio 40, rivi 4) UUF-tiedoston (Unigraf Update File) tiedostonimi, ja se päivittää kaikki laitekokoonpanoon liittyvät kortit yhdellä kertaa.

```
1   INT32 WINAPI UFGAPI_Device_UpdateFirmware(
2       INT32 iDeviceIndex,
3       UINT32 iFlags,
4       LPCSTR iFileName
5   );
```

KUVIO 40. Päivitysfunktion prototyyppi

Laitteen ohjelmistopäivitys on laiteohjauskirjaston kannalta monimutkainen prosessi. Ohjelmointirajapinnan käyttäjän kannalta päivitetään yksi laite, mutta todellisuudessa päivitetään kerralla kaikki kortit, jotka liittyvät kyseiseen laitteeseen. Päivitys kestää pisimmillään jopa 10 minuuttia korttia kohden, joten jokaiselle kortille luodaan päivitysajan optimoimiseksi oma säie (kuvio 41). Laitteen voi päivittää, vaikka se olisi suljettuna, koska päivitysfunktio avaa laitteen automaattisesti tarvittaessa. Avattuna olevan laitteenkin voi päivittää, mutta vain siinä tapauksessa, että sitä ei käytetä samanaikaisesti mihinkään muuhun. Automaattista avaamista ei ole kuvattu kuvion 41 sekvenssikaaviossa.



KUVIO 41. Ohjelmistopäivityksen sekvenssikaavio

7.6.4 Lähteen avaaminen ja sulkeminen

Lähes kaikki UFG-laiteohjauskirjaston toiminnot kohdistuvat kaapattavan datan lähteeseen. Tämän vuoksi rajapintaan ei toteutettu erikseen funktioita laitteen avaamiseen ja sulkemiseen, vaan ainoastaan lähteiden avaamiseen ja sulkemiseen. Laite avataan automaattisesti, kun ensimmäinen siihen liittyvä lähde avataan. Lähde avataan hakemalla sille kahva funktiolla UFGAPI_Device_GetSourceHandle (kuvio 42, rivit 1-6) ja suljetaan vapauttamalla kahva funktiolla UFGAPI_Device_ReleaseSourceHandle (kuvio 42, rivit 8-10). Kun viimeinen laitteeseen liittyvä kahva vapautetaan, kyseinen laite suljetaan automaattisesti.

Samalla kun lähde avataan, myös siihen liittyvä sisääntulo avataan. Laitteen rakenteesta johtuen vain yksi tulo voi olla kerrallaan auki, joten lähteiden kahvoja voi hakea vain yhteen tuloon kerrallaan. Ennen kuin toinen tulo voidaan avata, ensimmäisen tulon kaikki lähteet on suljettava. Muussa tapauksessa funktio palauttaa virheen.

Yhteen lähteeseen voidaan hakea useita kahvoja, ja lähdettä voidaan myös käyttää useista eri säikeistä. Kirjasto ylläpitää lähteelle viitelaskuria, joka kertoo, kuinka monta kahvaa lähteeseen on haettu. Kun kahva vapautetaan, viitelaskuri pienenee.

Viimeisen kahvan vapauttaminen sulkee lähteen. Viitelaskurin vuokaavio on esitetty kuviossa 21 sivulla 34. Viitelaskuri sisältää suojauksen, joten erillistä suojausta ei tarvita.

```
1   INT32 WINAPI UFGAPI_Device_GetSourceHandle(
2       INT32 iDeviceIndex,
3       INT32 iInputIndex,
4       INT32 iSourceIndex,
5       PINT32 oHandle
6   );
7
8   INT32 WINAPI UFGAPI_Device_ReleaseSourceHandle(
9       PINT32 ioHandle
10  );
```

KUVIO 42. Lähteen avaamis- ja sulkemisfunktioiden prototyypit

7.6.5 Tietueiden, asetusten ja tilojen käsittely

UFG-laitteet tuottavat paljon erilaisia mittaustietoja sisään tulevasta signaalista. Lisäksi niitä voidaan ohjata ja konfiguroida monipuolisesti. Useimmat tiedot ja asetukset ovat ryhmiteltävissä loogisiksi tietueiksi, mutta monikielisyystuen vuoksi tietueita ei voida suoraan välittää funktioiden parametreina. Tästä syystä rajapintaan kehitettiin mekanismi, jota kutsutaan nimellä Property Interface (ominaisuusrajapinta).

Tutkittaessa tietojen vaihtoa kirjaston, laitteen ja rajapintaa käyttävän ohjelman välillä huomattiin, että osa tiedoista voi olla suoria ja osa epäsuoria. Tämä tarkoitti sitä, että osa tiedoista voitiin viedä suoraan rajapintaa käyttävästä ohjelmasta laitteelle (tai kirjaston sisäiselle ohjaustoiminnolle) tai tuoda laitteelta, kun taas osa jouduttiin siirtämään kokonaisina ryhminä eli tietueina. Jälkimmäisessä tapauksessa yksittäiset tiedot piti välivarastoida kirjaston sisäiseen muistiin ja viedä sekä tuoda ryhmittäin.

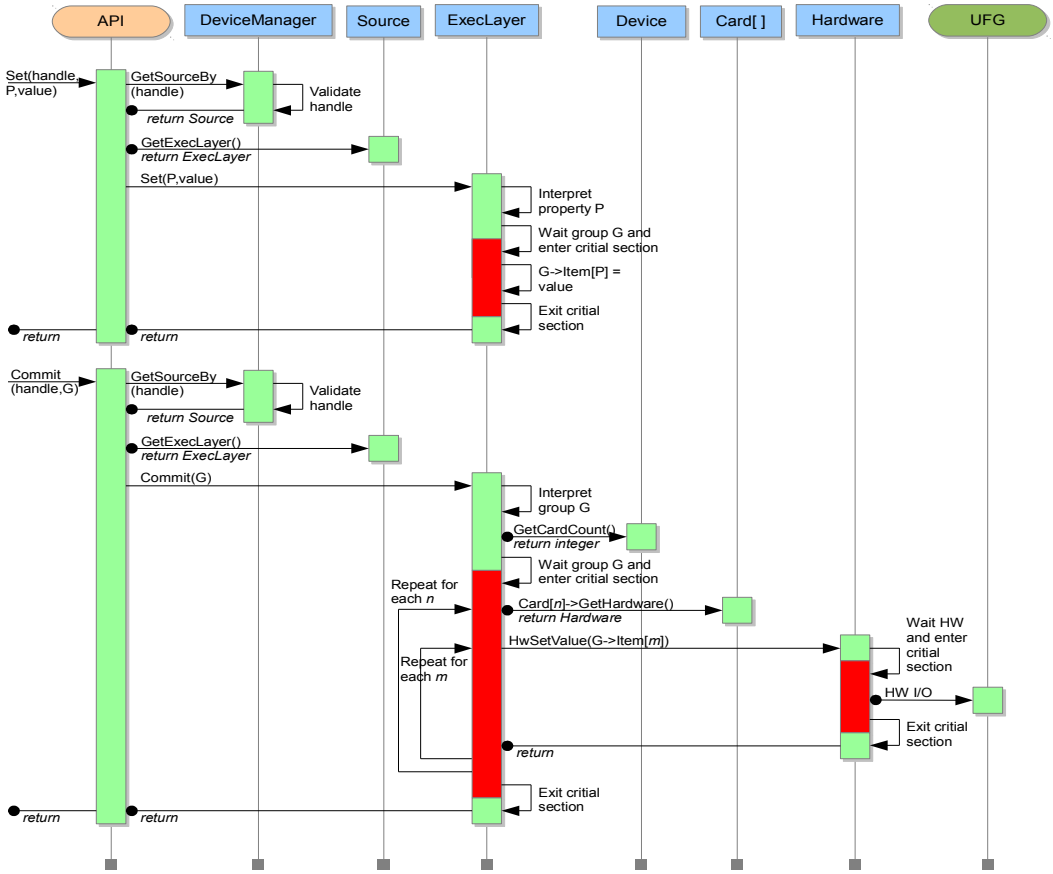
Property Interface toteutettiin neljällä funktiolla, joista kaksi siirtää kokonaisia tietueita laitteelta kirjaston sisäiseen muistiin ja muistista laitteelle, ja kaksi hakee yksittäisiä tietokenttiä muistista tai laitteelta käyttäjän ohjelmaan ja vie niitä muis-

tiin tai laitteelle. Toteutetut funktiot ovat UFGAPI_Property_Update, UFGAPI_Property_Commit. UFGAPI_Property_Get ja UFGAPI_Property_Set (kuvio 43). Näistä update- ja commit-funktiot käsittelevät tietueita ja get- ja set-funktiot yksittäisiä tietokenttiä.

Property Interface -toiminnot toteutettiin suorituskerroksen luokissa, joista ne välittyvät alempien tasojen kautta laitteelle. Suojaustarve analysoitiin piirtämällä sekvenssikaaviot kuvitteellisille käyttötapauksille. Näistä kaavioista esitellään epäsuoran tietueen *G* tietokentän *P* kirjoittaminen kuviossa 44. Muut kaaviot jätetään esittelemättä samankaltaisuuden vuoksi.

```
1   INT32 WINAPI UFGAPI_Property_Update(
2       INT32 iHandle,
3       INT32 iPropertyGroup
4   );
5
6   INT32 WINAPI UFGAPI_Property_Commit(
7       INT32 iHandle,
8       INT32 iPropertyGroup
9   );
10
11  INT32 WINAPI UFGAPI_Property_Get(
12      INT32 iHandle,
13      INT32 iProperty,
14      INT32 iSize,
15      PVOID oData
16  );
17
18  INT32 WINAPI UFGAPI_Property_Set(
19      INT32 iHandle,
20      INT32 iProperty,
21      INT32 iSize,
22      PVOID iData
23  );
```

KUVIO 43. Property Interface -funktioiden prototyypit



KUVIO 44. Tietueen *G* tietokentän *P* kirjoituksen sekvenssikaavio

7.6.6 Datan kaappaus

Tiedon kaappauksessa UFG-laitteelta tietokoneelle on kolme vaihetta: aloitus, tiedonkeruu ja lopetus. Näitä vaiheita varten rajapintaan toteutettiin funktiot UFGAPI_Capture_Start, UFGAPI_Capture_GetBlock ja UFGAPI_Capture_Stop, joiden prototyypit on esitelty kuviossa 45. Funktiot toteutettiin siten, että ne toimivat riippumatta kaapattavan datan tyypistä, eli samoilla funktioilla voidaan kaapata kaikista lähteistä.

Toteutuksessa otettiin huomioon, että dataa ei ole aina saatavilla. Jotta tiedonkeruufunktio ei jäisi jumiin, se toteutettiin ”pollaukseen” eli jatkuvaan kyselyyn perustuvana. Jos dataa ei ole saatavilla, funktio palauttaa kerätyn datan koon sijasta

nollan sekä lisäksi tilatiedon ”odottaa dataa”. Jos dataa on saatavilla, funktio palauttaa kerätyn datan koon.

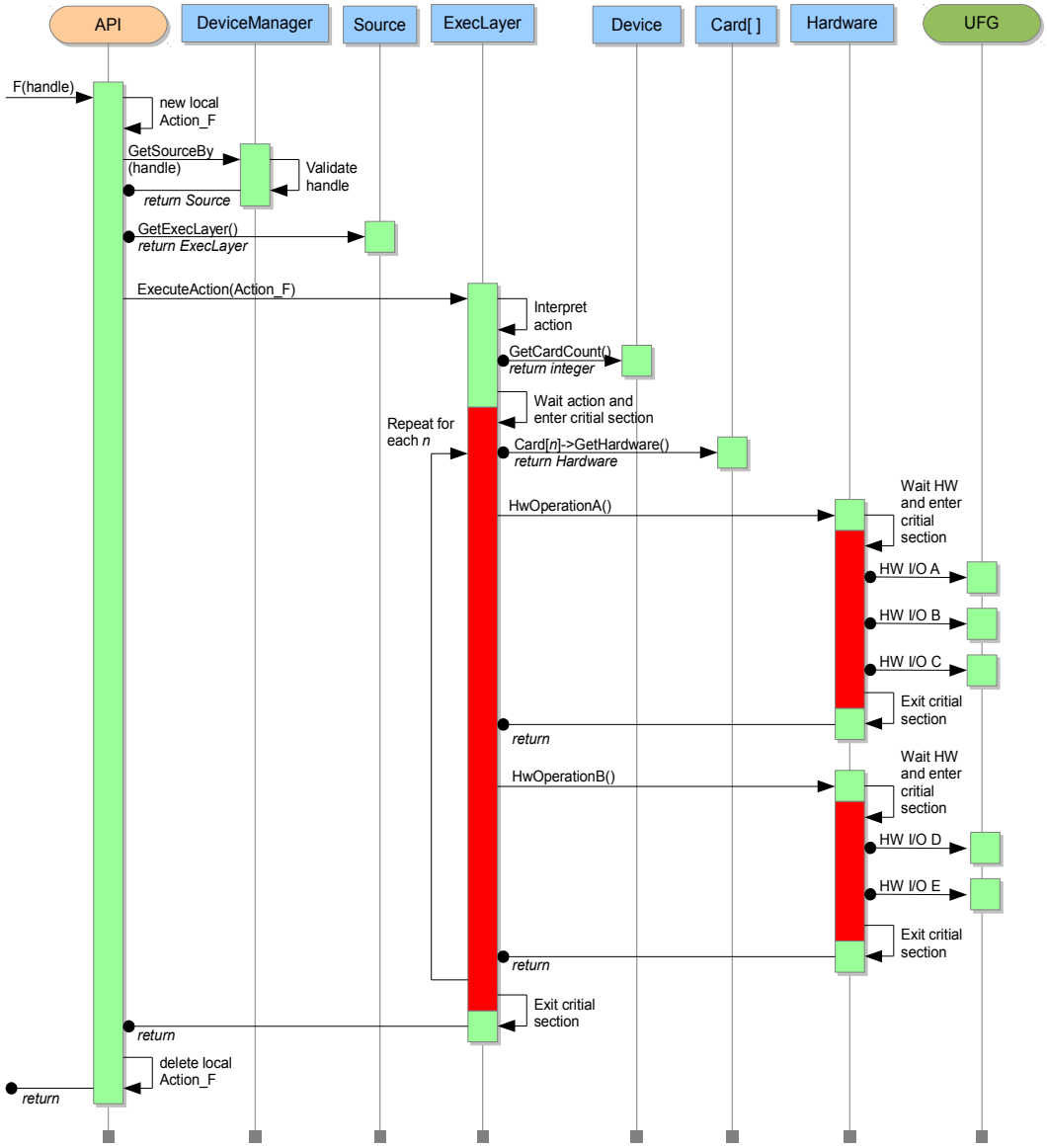
```
INT32 WINAPI UFGAPI_Capture_Start(
    INT32 iHandle
);

INT32 WINAPI UFGAPI_Capture_GetBlock(
    INT32 iHandle,
    INT32 iBufferSize,
    PVOID oBuffer,
    PUINT32 oStatus,
    PINT32 oTimeStamp
);

INT32 WINAPI UFGAPI_Capture_Stop(
    INT32 iHandle
);
```

KUVIO 45. Datankaappausfunktioiden prototyypit

Datankaappausfunktiot kuuluvat erikoistuneisiin funktioihin. Niille luodaan rajapinnassa action-objekti, joka välitetään suoraan suorituskerrokseen. Kuviossa 46 nähdään kuvitteellisen funktion *F* sekvenssi, joka pätee yleisiltä osiltaan kaikkiin erikoistuneisiin funktioihin. Siitä käy ilmi, miten erikoistuneiden funktioiden suojaus toteutettiin.



KUVIO 46. Kuvitteellisen funktiokutsun *F* sekvenssikaavio

7.6.7 HDMI- ja DP-liitäntöjen HPD-toiminto

HPD- eli Hot Plug Detect -toiminto on HDMI- ja DP-liitäntöjen ominaisuus ja se kertoo lähettävälle piirille, onko kaapeli kiinni laitteessa vai ei. Sitä voidaan ohjata UFG:llä keinotekoisesti tiloihin ”kaapeli kiinni” ja ”kaapeli irti”. Näin voidaan simuloida tapahtumaa, jossa kaapeli irtoaa, ja saada lähettävä laite alustamaan yhteys uudelleen.

HPD-toiminto oli alun perin tarkoitus toteuttaa käyttämällä Property Interfacea, mutta jossain vaiheessa se päätettiin kuitenkin toteuttaa erillisillä funktioilla. Tämä rikkoi alkuperäistä ajatusta pitää rajapinta mahdollisimman geneerisenä, mutta yhteensopivuuden takaamiseksi valittua menetelmää ei voinut enää myöhemmin vaihtaa. HPD-toimintoa varten laadittiin kolme funktiota, joilla voi tuottaa halutun mittaisen pulssin tai asettaa signaalin ylös tai alas.

Toteutetut funktiot ovat UFGAPI_HPD_Pulse, UFGAPI_HPD_High ja UFGAPI_HPD_Low (kuvio 47). Nämä funktiot kuuluvat kaappausfunktioiden tavoin erikoistuneisiin funktioihin.

```
1   INT32 WINAPI UFGAPI_HPD_Pulse(
2       INT32 iHandle,
3       INT32 iPulseWidth
4   );
5
6   INT32 WINAPI UFGAPI_HPD_High(
7       INT32 iHandle
8   );
9
10  INT32 WINAPI UFGAPI_HPD_Low(
11      INT32 iHandle
12  );
```

KUVIO 47. HPD-funktioiden prototyypit

7.6.8 Kuvien vertailu

Aikaisempiin kirjastoversioihin oli toivottu mahdollisuutta vertailla kahta kuvaa keskenään pikseli pikseliltä kuvavirheiden havaitsemiseksi. UFG-laite ei tue tällaista ominaisuutta laitteistotasolla, joten laiteohjauskirjastoon toteutettiin funktio UFGAPI_CompareFrame (kuvio 48), joka suorittaa vertailun ohjelmallisesti. Funktiota käytetään virheiden havainnointiin ja virhemäärien laskemiseen sekä virhekarttakuvan luomiseen. Se tukee kahta eri vertailumoodia, jotka asetetaan Property Interfacen kautta. Samoin virhelaskurit voidaan lukea ja nollata sekä maskeja asettaa Property Interfacen avulla. Funktio kuuluu erikoistuneisiin funktioihin.

```
INT32 WINAPI UFGAPI_CompareFrame(
    INT32 iHandle,
    INT32 iWidth,
    INT32 iHeight,
    PVOID iReferenceFrame,
    PVOID iCompareToFrame,
    PVOID oResultFrame
);
```

KUVIO 48. Vertailufunktion prototyyppi

7.6.9 Käyttöliittymän apufunktiot

Numeeristen virhekoodien ja erilaisten tyyppikoodien tulkinta käyttöliittymissä havaittiin hankalaksi. Kirjastoon toteutettiin apufunktioita, joilla voidaan hakea tekstimuotoisia englanninkielisiä kuvauksia numeerisille arvoille. Muita kieliä käyttävien on edelleen tulkattava numeeriset arvot itse tai luotava vastaava funktio omalle kielelle. Kaikki apufunktiot toteutettiin siten, että ne ovat vapaakäyntisiä ja toimivat riippumatta kirjaston alustuksesta.

Laitteiden, sisääntulojen ja lähteiden tyyppikoodeja varten toteutettiin funktiot UFGAPI_Device_GetDeviceName, UFGAPI_Device_GetInputName ja UFGAPI_Device_GetSourceName (kuvio 49). Virhekoodeja varten puolestaan toteutettiin funktio UFGAPI_VerboseError (kuvio 50).

```
INT32 WINAPI UFGAPI_Device_GetDeviceName(
    INT32 iDeviceId,
    LPCSTR oDeviceName
);

INT32 WINAPI UFGAPI_Device_GetInputName(
    INT32 iInputType,
    LPCSTR oInputName
);

INT32 WINAPI UFGAPI_Device_GetSourceName(
    INT32 iSourceType,
    LPCSTR oSourceName
);
```

KUVIO 49. Tyyppikoodien kuvauksenhakufunktioiden prototyypit

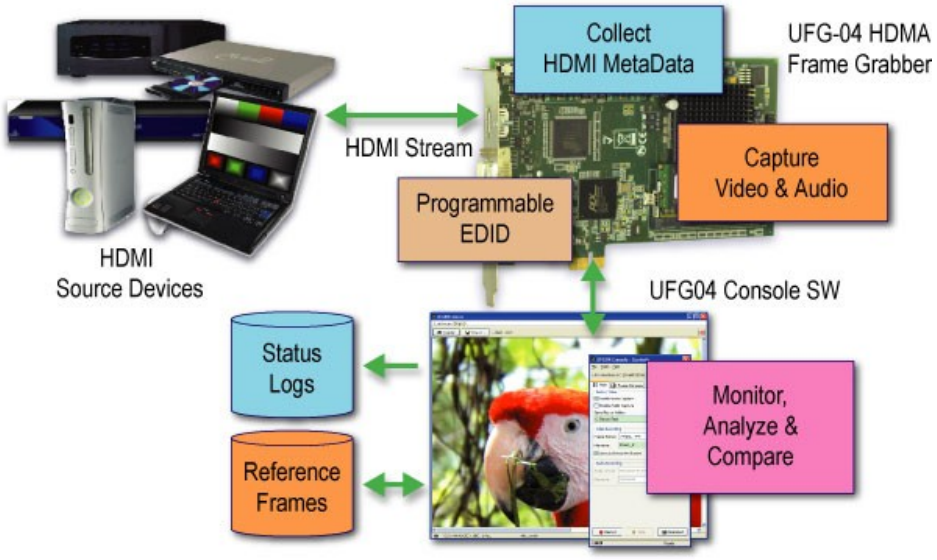
```
1   INT32 WINAPI UFGAPI_VerboseError(
2       INT32 iErrorCode,
3       LPCSTR oErrorStr
4   );
```

KUVIO 50. Virhekoodin kuvauksenhakufunktion prototyyppi

7.7 Kirjaston testaus

Kirjastoa testattiin Unigrafin toimesta sovelluskehityspaketin sisältämillä yksisäikeisillä esimerkkiohjelmilla ja monisäikeisellä UFG Console -kuvankaappausohjelmalla. Nämä kaikki käyttävät opinnäytetyössä toteutettua laiteohjauskirjastoa UFG-laitteiden ohjaukseen.

Testikokoonpanossa käytettiin kuva- ja äänilähteenä useita eri laitteita, esimerkiksi Unigrafin omaa VTG-tuotetta, kolmannen osapuolen video-, DVD- ja BlueRay-soittimia sekä pelikonsoleja (kuvio 51). Testiympäristönä käytettiin useita eri tietokoneita, joissa oli asennettuna laiteohjauskirjaston tukemat Windows-käyttöjärjestelmäversiot. Kaikki UFG-laitteiden tukemat sisääntulostandardit testattiin erikseen.



KUVIO 51. Esimerkki UFG-laitteen testiympäristöstä

Laiteohjauskirjastoa testattiin sekä useita päiviä kestäneillä jatkuva-aikaisilla kaappaustesteillä että eri asetuksia vaihtelevilla lyhytaikaisilla testeillä. Lisäksi testattiin laitteiden käyttöä useasta eri ohjelmasta yhtä aikaa. Kirjasto toimi kaikissa testitilanteissa halutulla tavalla ja vakaasti ilman kaatumisia, jumiutumisia tai tietojen vahingoittumisia. Testauksen aikana havaittiin kuitenkin joitakin ongelmakohtia, joissa valitut testiohjelmat käyttäytyivät vakaasti, mutta jotka saattavat ääritapauksissa johtaa odottamattomiin tilanteisiin. Tällaisia olivat esimerkiksi jotkin kirjaston rajapinnan käyttäjän tekemät ohjelmointivirheet, joita ei ollut otettu huomioon.

Testauksen aikana havaittiin myös tarvetta kaappausfunktioiden optimoinnille. Lisäksi löydettiin muitakin rakenteellisia ja toiminnallisia parannusehdotuksia. Näiden tapausten analysointi ja toteutus päätettiin rajata opinnäytetyön ulkopuolelle ja jättää jatkokehitystyön aiheeksi.

8 YHTEENVETO

Opinnäytetyön tavoitteena oli toteuttaa säieturvallinen oliopohjainen laiteohjauskirjasto. Työssä pyrittiin vastaamaan kysymykseen, miten oliopohjaisella toteutustavalla ja Windows-käyttöjärjestelmän tarjoamilla suojausmenetelmillä voidaan suojata laiteresursseja ja ohjelman tiloja moniajon aiheuttamilta ongelmilta DLL-kirjastossa, jossa funktioiden kutsujärjestys ei ole etukäteen tunnettu.

Työn teoriaosuus kattoi aihepiirin hyvin, ja sen avulla toteutettiin laiteohjauskirjasto, joka osoittautui testeissä säieturvalliseksi. Kirjaston toteutuksessa täytettiin kaikki ennalta asetetut tavoitteet ja vaatimukset. Myös työn aikana kohdatut ongelmat saatiin ratkaistua, ja tuloksena oli luotettava laiteohjauskirjasto, joka voitiin julkaista asiakkaille.

Opinnäytetyössä toteutettu laiteohjauskirjasto on liitetty osaksi UFG Software Package -ohjelmistoa ja siitä on jo saatu runsaasti käyttökokemuksia ja palautetta asiakkailta. Kirjasto on osoittautunut helppokäyttöiseksi ja mukautuvaksi, ja asiakkaat ovat olleet tyytyväisiä sen luotettavuuteen ja joustavuuteen. Property Interface -rajapinnan omaksuminen on tuottanut joillekin vaikeuksia, mutta tällaisia ongelmia on vähentänyt kirjastolle laadittu perusteellinen käyttöohje. Kirjaston kehitys jatkuu aktiivisesti laitteiden kehittyessä ja uusia laitteita suunniteltaessa.

Työn aikana havaittiin joitakin suunnittelupuutteita, jotka tulisi ottaa huomioon jatkokehityksen aikana. Esimerkiksi viitelaskurijärjestelmän laskurin ylärajaa ei ole suojattu, vaan laskuria voi kasvattaa loputtomiin, kunnes se vuotaa yli. Tämän voisi korjata rajoittamalla laskuria semaforilla. Suorituskerroksen luokissa tulisi soveltaa moniperintää, mikä saattaisi vähentää tarvetta saman koodin uudelleenkirjoittamiseen. Lisäksi VCL-kirjastosta ja kääntäjäkohtaisista erityispiirteistä tulisi pyrkiä eroon, jotta kirjastosta voitaisiin kehittää alustariippumaton.

LÄHTEET

Amdahl, G. 1951. The logical design of an intermediate speed digital computer. Wisconsin: University of Wisconsin.

Cooling, N. 2011a. Mutex vs. Semaphores – Part 1: Semaphores [viitattu 15.10.2011]. Saatavissa: http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%E2%80%93-part-1-semaphores/

Cooling, N. 2011b. Mutex vs. Semaphores – Part 2: The mutex [viitattu 15.10.2011]. Saatavissa: http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%E2%80%93-part-2-the-mutex/

IBM. 2011. The S/360: A turning point in mainframe history [viitattu 12.10.2011]. Saatavissa: http://publib.boulder.ibm.com/infocenter/zos/basics/topic/com.ibm.zos.zmainframe/zconc_s360history.htm

Labrosse, J. 2002. MicroC/OS-II, The Real-Time Kernel, Second edition, San Francisco: CMP Media LLC.

LinuxManPages. 1995. FORK [viitattu 15.10.2011]. Saatavissa: http://linuxmanpages.com/man2/fork.2.php

MSDN. 2011a. CloseHandle Function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms724211(VS.85).aspx

MSDN. 2011b. CreateMutex function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms682411(VS.85).aspx

MSDN. 2011c. CreateProcess Function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms682425(VS.85).aspx

MSDN. 2011d. Critial Section Objects [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530(v=vs.85).aspx

MSDN. 2011e. Introduction to Spin Locks [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ff548114.aspx

MSDN. 2011f. ReleaseMutex Function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms685066(VS.85).aspx

MSDN. 2011g. Threads and Processes [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms164740(v=vs.80).aspx

MSDN. 2011h. WaitForMultipleObjects Function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms687025(v=VS.85).aspx

MSDN. 2011i. WaitForSingleObject Function [viitattu 15.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms687032(VS.85).aspx

MSDN. 2011j. _spawn, _wspawn Functions [viitattu 20.10.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/20y988d2(v=vs.71).aspx

Olsson, G. & Piani, G. 1992. Computer systems for automation and control. Hemel Hempstead: Prentice Hall International (UK) Ltd.

Oney, W. 2002. Programming the Microsoft Windows Driver Model. Second edition. Redmond, Washington: Microsoft Press.

Silberschatz, A., Peterson, J. & Galvin, P. 1991. Operating System Concepts. Third edition. Massachusetts: Addison-Wesley Publishing Company, Inc.